Self-Testing and Self-Repairing Embedded Processors: Techniques for Statically Scheduled Superscalar Architectures

By the Faculty of Mathematics, Natural Sciences and Computer Science of the Brandenburg University of Technology Cottbus-Senftenberg

on July 3rd 2014 accepted Habilitation Thesis

to gain the degree

Doctor rerum naturalium habilitatus (Dr. rer. nat. habil.).

Submitted on February 2nd 2014 by

Dr. rer. nat. Mario Schölzel

born on November 17th 1975 in Lübben.

Reviewers

Prof. Dr.-Ing. Heinrich Theodor Vierhaus (BTU Cottbus-Senftenberg, Germany)

Prof. Dr. sc. techn. Raimund Ubar (TU Tallinn, Estonia)

Prof. Dr. Matteo Sonza Reorda (Politecnico di Torino, Italy)

Acknowledgment

First of all I would like to thank Professor Vierhaus for offering the chance to write my habilitation thesis and his tireless networking, which provided the base for building excellent international relationships.

Special thanks also to all my colleagues, in particular Tobias Koal for many fruitful discussions, to Stefan Scharoba and Sebastian Müller for their contributions in my field of research, and to Kathleen Galke for organizing the social life and events in the computer engineering group and, of course, for preparing the coffee.

Last but not least very warm thanks to Susann, Mara and Malia just for being there and making life beautiful.

Content

Chapter 2

Chapter 3

Preface

This thesis introduces a comprehensive approach for making a particular class of embedded processors self-testing and self-repairing, such that a limited amount of permanent hardware faults that occur during the lifetime of these processors in the field will not prohibit the functional behavior of the user application running on the processor. The presented concepts all use redundant hardware, but the techniques used for administrating the hardware-redundancy range from hardware-based methods over hybrid methods to pure software-based methods, whereby the focus is on the latter ones. The proposed methods will be demonstrated by using a processor that is well designed for diagnostic self-test and self-repair purposes. This will also highlight some architectural properties of such a processor, which are beneficial for performing a software-based self-test and self-repair process.

Chapter 1 is an introduction to the field of dependable systems and fault tolerance. Fundamental terms and notations, which are used throughout this thesis for classification and evaluation, are provided. The used processor model – the VARP processor – is introduced in chapter 2 together with a hardware-based self-repair scheme for that processor. The results are used as reference values for evaluating the software-based methods. Chapter 3 introduces the fundamental concept of the software-based self-repair. In chapter 4 hybrid methods are derived by combining software-based and hardware-based methods, highlighting the synergy effects of the combination. Finally, in chapter 5, a diagnostic and adaptive software-based self-test scheme is introduced. This self-test scheme provides the diagnostic capability that is needed in the field for identifying defect components in the VARP processor and completes the comprehensive software-based self-test and self-repair approach. Each chapter provides a short summary of the advantages and disadvantages of the presented methods. Chapter 5 will close with a summary for the comprehensive approach.

Chapter 1

Introduction and Foundations

The shrinking feature size of integrated circuits described by Moore's law is twofold. On the one hand it allows for building integrated circuits that can perform more complex tasks much faster, at lower power consumption, and at a cheaper price per transistor and per function. For those reasons integrated circuits get into more and more fields of our everyday life. They are embedded for special purpose computing into consumer electronics like mobile phones, audio devices and TVs, but also into mechanical systems like aircrafts, cars, and medical devices[1]. There is a clear trend that more and more functionality in electronic and mechanical devices will be controlled by embedded systems, and the functionality that is provided by them becomes more and more complex. This requires very powerful embedded systems that can be obtained, for example, by further shrinking the feature size[2]. On the other hand, the shrinking feature size of integrated circuits increases their vulnerability to various physical defects [32, 190] leading to a reduced life time [176]. As a consequence, it becomes more likely that they can compute wrong results or even totally give up their functionality during their operational phase [49]. Such a malfunction in an embedded system for consumer electronics may be annoying, but acceptable. However, for safety- and mission-critical embedded systems that are used for example in automotives or

[1] It is common to refer to these embedded integrated circuits for special purpose computing as *embedded systems*.

[2] Current CMOS process from Intel for processor manufacturing uses 22 nm since 2011 and will move to 14 nm in 2014 [78].

avionics, a malfunction may become life-threatening for persons or will cause the loss of high-priced equipment.

Automotive applications are a good example for this trend. A few years ago integrated circuits were used in cars only for controlling relatively uncritical and simple functions like window opener, engine injection control, etc. These functions did not require extraordinary computational performance. Thus, the integrated circuits could be manufactured using technologies with larger feature size that were very robust. These devices had a long lifetime. Reliability of hardware was not a major concern, except for contacts exposed to corrosion. But this situation now changes. More complex driver assistance systems become state of the art for cars, finally maybe reaching a point where cars drive completely autonomously. This makes the safety of car passengers much more dependent on the reliability of the embedded electronic systems. Many functions of these systems require complex and computational expensive image and signal processing that must be performed in real-time. This performance may be only achieved with complex hardware using powerful processors probably containing hundreds of millions of transistors. This requires the manufacturing of these processors with nano-scaled feature sizes that makes them, unfortunately, less reliable; i.e., they are more vulnerable to various physical defects during their lifetime.

Therefore, solutions are needed that allow for building dependable embedded systems and processors, even when the nano-scaled hardware becomes less reliable. I.e., already the development process of the processor must take into account that some defects will occur during the operational phase. In order to make sure that the processor can continuously provide the desired functionality even in the presence of some defects, fault tolerance techniques like self-test and self-repair capabilities can be included into the processor. These techniques enable the processor to handle occurring physical defects autonomously in the field by some kind of reconfiguration. This targeted scenario is shown in figure 1-1. After manufacturing and test, the processor is in use and runs a user application that uses the computational resources of the processor, for example *C1* to *C6*. When a defect affects a particular component of the processor, for example *C3*, then the user application is interrupted. A self-test is performed in order to localize the defective component. The self-repair is used for reconfiguring the processor, such that the defective component is not used anymore. If this reconfiguration is successful, then the execution of the user application is resumed, for example by a restart of the system. Please note that such a scenario also covers situations where the misbehavior of a defect component is masked for some time by appropriate fault tolerance techniques without interrupting the user application. Then, the user application is interrupted some time later, when there is enough time for performing the time consuming self-test and self-repair functionality.

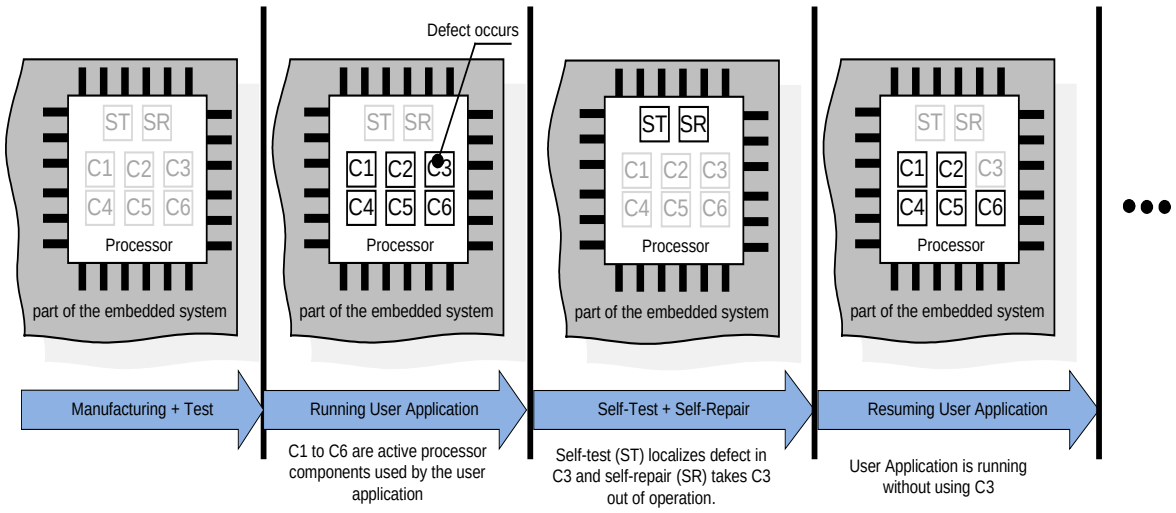


Figure 1-1: Targeted autonomous self-test and self-repair scenario.

The scenario in figure 1-1 takes consecutive defects in multiple processor components into account. I.e., a processor that has been successfully reconfigured due to an occurred defect can handle another defect that occurs some time later, provided that there are sufficient computational resources available. The focus of this thesis is on software-based methods for providing the self-repair and self-test capability in such a processor and in such a scenario. Thereby the capability of handling defects in multiple small components of the processor allows for an efficient usage of the computational resources in the presence of multiple defects, because the extent of a defect can be limited in most situations to a small component containing the defect.

1.1 Dependable and Reliable Systems

First, some fundamental definitions for various terms, which are related with dependable systems and used throughout this thesis, are provided. According to [15] the dependability concept first appears in the 1830 in the context of Babbage's Calculating Engine, where it was first stated that

> *"The most certain and effectual check upon errors which arise in the process of computation, is to cause the same computations to be made by separate and independent computers; and this check is rendered still more decisive if they make their computations by different methods"* [16].

In this text the problem of trusting in the result of a computation is addressed. Exactly the same issue is addressed by the term *Dependability*, whose definition has evolved over the past decades. A condensed definition, which can be also found in a similar way in many textbooks and papers, is given by Laprie [103]:

Definition 1-1 (Dependability):
Dependability is defined as that property of a computer system such that reliance can justifiably be placed on the service it delivers. The service delivered by a system is its behavior as it is perceptible by its user(s); a user is another system (human or physical) which interacts with the former.

In other words, the dependability of a system depends on how much the user trusts in the correctness of the output (or produced data) of the system. In many situations this also implies that it is expected that the system continuously delivers an output within a certain time. According to [14] there are three issues that will affect the trust in the correctness and continuity of the outputs. They are shown as the first level of the dependability tree in figure 1-2.

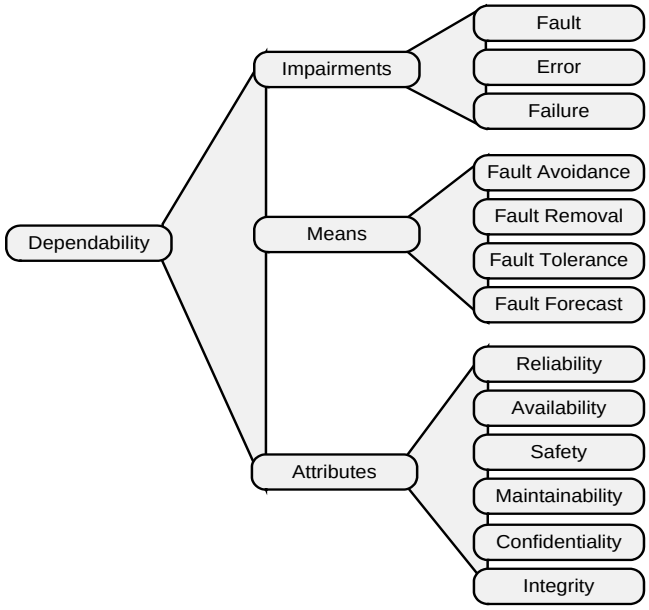


Figure 1-2: Dependability tree adopted from [102] and [140].

The confidence in the correctness of the output is reduced by *Impairments*. These are faults, errors, and failures, which have a negative impact on the trustworthiness. On the other hand, the confidence is improved, if it is known that the system includes methods for handling faults or it has been developed by using methods to avoid certain types of faults. These *means* have a positive impact on the dependability, and they can be used to overcome the negative impact of impairments. For example, a software program that has been formally verified is more trusted than a software program that has been tested only. Finally, the *attributes* are measures to quantify the positive and/or negative impact of means and/or impairments. Their usage will also contribute to the confidence that is put on the system, because, for example, they allow for comparing different versions of a system. Based on this comparison it can be decided which solution is trusted

more. The next three subsections will give clear definitions for impairments, means, and attributes.

1.1.1 Faults, Errors, and Failures

This section reviews the relationship between faults, errors, and failures, which is a cause-effect relationship. I.e., a fault may cause an error, and an error may cause a failure. This relationship requires the system to be considered as a hierarchically organized one. I.e., there is a top-level system, which is composed of various sub-systems, and each sub-system may be composed of zero or more sub-systems again.

1.1.1.1 Failures

Definition 1-2 (Failure, adopted from [102]):
A *failure* occurs when the delivered service of the top-level system deviates from the correct service.

I.e., some of the functionality of the system is performed incorrectly, and this can be observed at the outputs of the top-level system. This may affect the behavior of the user of the top-level system. Thereby, the user may be another system. Please note that the definition does not refer to the specification of the system. Therefore, it also covers mistakes in the specification or missing aspects in the specification. According to the cause-effect relationship, a failure is the effect of an error.

1.1.1.2 Errors

Definition 1-3 (Error):
An *error* occurs in a system, if at least one of its sub-systems has a failure.

Thus, an error can be seen as incorrect data or behavior that can be observed inside a system, but the incorrect data or behavior may be not observable outside of the system. I.e., the system does not necessarily have a failure. An error is either *latent* or *detected* [14]. An error is detected, if the system is aware of its presence. It is latent if it is present, but not detected, yet. An error may propagate inside the system from one sub-component to another, finally maybe appearing at an output of the top-level system. Thus, an error may be the cause of other errors or of a failure. Please note that an error in a (sub-) system may be introduced from outside, i.e., the system receives erroneous inputs. According to the cause-effect relationship, an error may be the effect of a fault.

1.1.1.3 Faults

In the literature various meanings for the term *fault* can be found. A very general definition that is used in the fault tolerance community is given in [14].

Definition 1-4 (Fault, from [14]):
A fault is the adjudged or hypothesized cause of an error.

This definition also covers the situation that the fault is located outside the system boundaries. Such faults are called *external faults*. For example, some kind of radiation interacts with a system and changes the state of a memory cell. The fault (the radiation) has caused an error (the wrong value in the memory cell). *Internal faults*, on the other hand, originate from inside the system boundaries. For example, a manufacturing flaw of the device is an internal fault that may cause an error. Thereby a fault is called *active* when it causes an error, otherwise it is called *dormant* [14]. These situations, where faults are originated only inside the system, are also covered by another widely used definition from the fault tolerance community.

Definition 1-5 (Fault, from [140]):
A fault is a physical defect, imperfection, or flaw that occurs within some hardware or software component.

In this definition the fault is clearly located inside the system, because it is stated that the fault is located either in a hardware or software component of the system. Moreover, according to definition 1-5 each physical defect is considered as a fault. However, in the test community there is usually a difference between a defect and a fault, which is expressed by definition 1-6.

Definition 1-6 (Fault, from [187]):
A fault is a representation of a defect reflecting a physical condition that causes a circuit to fail to perform in a required manner.

The difference is that in definition 1-5 a (physical) defect (either a hardware defect or software defect) is the same as a fault, while in definition 1-6 a fault is an abstracted representation of a physical hardware defect. For example, consider a NAND-gate whose output is connected to ground, due to some manufacturing flaw. This physical defect is already a fault, according to definition 1-5. However, in an abstracted representation of the circuitry, for example as net-list at gate-level, it is not possible to reflect accurately all aspects of the physical defect. Therefore, an abstracted representation of the defect is needed for representing it in the model of the circuitry. This abstraction is provided by the *fault model*. The objective of a fault model is to reflect the behavior of physical defects as accurately as possible for test purposes, while maintaining the computational complexity, for example for fault simulation and test pattern generation, as low as possible [187]. For this reason a fault model specifies:

- all possible *sites* for faults in the model of the circuitry and

- all *states* of a fault.

The probably most widely used fault model for decades at logic level is the single *stuck-at fault model* [53]. Possible sites of faults in that model are primary output and input signals, internal gate inputs and outputs, fan-out stems, and fan-out branches. Only a single fault is allowed in the circuitry model, when the single stuck-at model is used. Each fault has either the state *stuck-at-0* (*SA0*) or *stuck-at-1* (*SA1*), meaning that the corresponding signal has always the logic value 0 (SA0) or the logic value 1 (SA1). The stuck-at fault model is used for the development of the self-test presented in chapter 5. Many other fault models exist for reflecting more accurately other aspects of a defect at various abstraction layers [128, 187]. However, for fault handling in reliable systems it is not necessary to consider a fault as a representation of a defect at a particular abstraction layer. Rather, particular properties of the defect are important, which remain the same even at different abstraction layers. They are introduced in the next section. Hence, the terms fault and defect are used interchangeably in the context of fault handling.

1.1.1.4 Fault Classes

The presented self-repair methods in this thesis do not rely on a specific fault model. But they rely on specific properties of faults. Eight viewpoints for characterizing all possible faults that may affect a system have been presented in [14]. These viewpoints are listed as *elementary classes* in figure 1-3.

	Elementary class	Property	Description
Fault Classification	Phase of creation or occurrence	Development faults	Occur during system development
		Operational faults	Occur during operational phase
	Persistence	Permanent faults	Presence is continuous in time
		Temporary faults	Presence is bounded in time
	System boundaries	Internal faults	Originate inside system boundaries
		External faults	Originate outside system boundaries
	Dimension	Hardware faults	originate in or affect hardware
		Software faults	affect software (program and data)
	Phenomenological cause	Natural faults	Caused by natural phenomena
		Human-Made faults	Result from human actions
	Intent	Deliberate faults	Result of a harmful decision
		Non-Deliberate faults	Introduced without awareness
	Capability (of the developer)	Accidental faults	Introduced inadvertently
		Incompetence faults	Result from lack of professional competence
	Objective	Malicious faults	Introduced with objective of causing harm
		Non-Malicious faults	Introduced without malicious objectives

Figure 1-3: Classification of faults taken from [14].

Each elementary class uses two mutual exclusive properties for the characterization of a fault, such that each fault is characterized by a combination of eight properties from eight different elementary classes. Thereby not all of the 256 possible combinations are meaningful. In [14] 31 meaningful combinations were identified, and only five of them are of particular interest for this thesis. These are exactly those faults that have a *natural cause*; i.e., they are not human-made. Therefore, they are also non-deliberate (*intent*), accidently introduced (*capability*), and non-malicious (*objective*). Moreover, natural faults will be always hardware faults (*dimension*), because software faults are human-made. According to the remaining elementary classes, natural faults can be further distinguished as shown by the tree in figure 1-4.

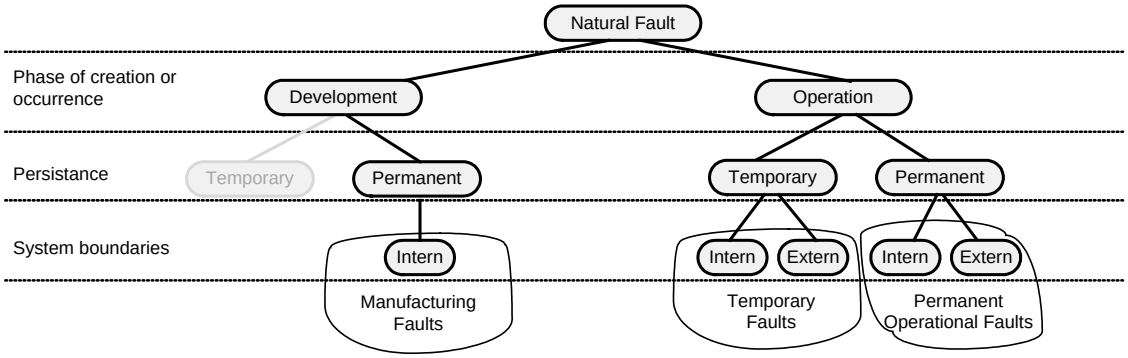


Figure 1-4: Grouping of fault classes to manufacturing faults, temporary faults and permanent faults.

According to their phase of creation, natural faults are introduced either during the manufacturing phase (*development faults*) or when the system is in operation (*operational faults*). The persistence of development faults is always permanent, while operational faults can be of temporary or permanent persistence. The *persistance* of a fault characterizes the time that a fault is present in the system. Permanent faults remain in existence indefinitely long [140]. Temporary faults will be present only for a short time period. After that time period they disappear. If the temporary fault is an external fault, then it is often called a *transient fault*. These are typically faults that are caused by some kind of external radiation or electromagnetic interference. Internal temporary faults are often named as *intermittent faults*. Intermittent faults are also faults with a short duration, but, in contrast to transient faults, they appear repeatedly. In many cases they are caused by some kind of manufacturing flaw or aging effect in combination with a particular internal system state or environmental conditions, e.g. the temperature of the system.

Each leaf of the tree figure 1-4 represents a combined fault class. They are grouped into

- Manufacturing faults,
- Temporary faults, and

- Permanent operational faults.

It can be noticed that manufacturing faults and permanent operational faults have almost the same properties, except for the phase of their creation. Therefore, they are grouped together to the class of *permanent faults*. Although the self-repair methods presented in this thesis will primarily target for permanent operational faults, they can be also used for handling manufacturing faults. Then, it may become possible to improve the yield or to relax the demands for production testing as it is claimed in the ITRS roadmap [3] for the future.

1.1.1.5 Natural Causes of Faults

This section will briefly review some natural causes for temporary and permanent faults.

Causes for Manufacturing Faults

Faults belonging to the group of manufacturing faults are introduced during the development phase of the system. They do affect hardware components only and their persistence is permanent. They are created by flaws and variations during the manufacturing process of the integrated circuit, including:

- pollutions of the waver,
- crystal imperfections in the bulk,
- missing contacts and shorts due to flaws during the metallization process,
- oxide defects resulting in oxide shorts and gate oxide pinholes,
- imperfections due to lithography problems and
- improper doping in the transistor channel due to stochastically effects [171].

Especially stochastically effects play an important role for manufacturing nano-scaled circuits, because they are hardly controllable during the manufacturing process. By such effects the distribution of device parameters becomes wider. For example, the *random dopant fluctuation* is a deviation of the expected doping density in the channel area of a field-effect transistor. In 40 nm devices no more than 100 doped atoms are expected along the channel of the transistor [114]. Even a small deviation in the doping profile will change significantly the threshold voltage of the transistor, which can have a significant impact on the delay of the logic circuit [114]. Also in Intels FinFET technology, which is used in 22 nm technology and below, the *random dopant fluctuation* has a strong impact [44]. The exhibited misbehavior of the circuit is a *delay fault*, i.e. the correct signal arrives, but not in time. Because such statistical variations during the manufacturing process cannot be avoided, the circuits are designed with timing and voltage margins [12]. These margins will degrade the achievable performance

of the circuit. But reducing these margins may cause more devices to fail due to delay faults. Including fault tolerance techniques in devices with reduced margins, may increase the yield at the cost of a degraded performance of only some devices. An example for such an approach is the *RAZOR-latch* [56]. The goal is to eliminate the need for voltage margins by correcting delay faults that will occur in some processors at low supply voltage levels when dynamic voltage scaling is used for power-saving reasons. The RAZOR-latch allows for detecting delay faults during runtime, such that the fault can be handled in those devices that have a slow path due to some parameter variations.

Causes for Temporary Faults

External temporary faults are caused by single event effects like energetic particles that hit the circuitry or by other electrical sources that do not permanently damage the circuitry. Energetic particles include alpha particles [68], protons [127] and neutrons [68, 72]. Faults from electrical sources include electromagnetic interference, power supply noise and radiation [189]. When the fault flips the value of a memory element, then it is called a *single-event-upset* (*SEU*). Special techniques for masking SEUs have been developed [122, 173]. When the fault affects a signal in the combinatorial logic, then it is called a *single-event-transient* (*SET*). The erroneous signal in the combinatorial circuitry may be not latched or it is turned back into the correct signal by the driver of the gate or wire. Special techniques were developed for protecting memory elements from latching SETs [122]. Errors induced by SEUs and SETs are also called *soft-errors*. For 65 nm technology it is reported that only 11% of the soft-errors affect the combinatorial part of typical microprocessors. 49% will affect the sequential parts of the circuitry and 40% unprotected SRAMs [122]. Thereby, soft-errors make up the biggest portion of operational faults compared with permanent faults. The portion ranges from 75% to 99.9% [139]. This ratio depends at least on the environmental and operational conditions of the system. E.g., a processor that is operated in high altitudes will have higher probabilities for transient faults than the same processor at sea level. For this reason the ratio between transient and permanent faults will differ for these two processors, assuming a constant probability for permanent faults.

Causes for Permanent Faults

Of particular interest for this thesis are permanent faults that arise during the operational phase of the processor. These permanent faults either have an internal or an external origin (see figure 1-4). External permanent faults are caused by a single-event effect that causes a permanent damaging of the system. Well known is the *single-event latch-up* (*SEL*). This effect is caused by the passage of an energetic particle through a sensitive region of a transistor, which induces parasitic effects

that create a short between power and ground. The high current flow locally creates high temperatures that may cause local metal melting and destroys the local structures. However, not in every case the local structure is destroyed, but, at least the power supply must be interrupted in order to resume to normal operation [51]. Another single-event effect is caused by energetic ions that will cause a single-event gate rupture [131]. This effect permanently damages the gate insulator layer of a transistor, such that the current flow from source to drain is no longer controllable [154].

There are also various reasons for the occurrence of internal permanent faults in nano-scaled CMOS circuits during their operational phase. They are subsumed as *aging effects*. *Hot carrier injection* (*HCI*) and *negative bias temperature instability* (*NBTI*) are non-destructive aging effects. HCI is an effect where electrons are highly accelerated by the electrical field in the channel of a n-channel field effect transistor (*FET*). They undergo impact ionization, which generates electron-hole pairs. The carrier of the higher energy (hot carrier) becomes trapped in the gate oxide film of the FET [57, 99]. By accumulating hot carriers in the gate oxide film, the threshold voltage of the transistor deteriorate, which finally makes the transistor slower. NBTI occurs in p-channel FETs, when a negative gate bias is applied [8]. Holes from the silicon surface of the channel are trapped in the interface of the silicon and the gate oxide film. There, the holes disassociate hydrogen atoms from the silicon atoms. The hydrogen ions are caught in the gate oxide film and cause a positive charge there [99]. This also changes the threshold voltage of the transistors and the transistor becomes slower, too. The occurrence of both effects in a processor will not destroy the affected transistors. But both effects may cause delay faults. Delay margins may be introduced during the system design in order to avoid the activation of a delay fault[3]. However, the delay from aging may accumulate with other delay effects, e.g., caused by the random dopant fluctuation effect. This requires either very large worst-case margins resulting in non-optimal performance yield for most processors, or methods that act dynamically; e.g., a dynamic frequency scaling of the processor with respect to aging effects as proposed in [117].

Beside degradation effects like HCI and NBTI, there are also destructive effects like *time-dependent dielectric breakdown* (*TDDB*) and *electro migration* (*EM*) that will cause permanent faults. TDDB is a degradation of some insulator layer within the circuitry. This can be either a thin gate oxide film of a field effect transistor or an insulator material between wires. In the latter case the resistance between both

[3] A delay fault is activated, when the correct signal arrives too late to be latched in a memory element.

wires is decreased, which may cause a short [1]. When TDDB affects the dielectric film between the gate and the channel of a FET, then the transistor is no longer controllable by the gate [18]. The failure mechanism can be partly controlled by process parameters, such that it will not appear within the expected operational life time of the system. However, variations during the manufacturing process (e.g., due to stochastical effects) may produce disturbances in the gate oxide films, which will accelerate the TDDB effect, finally yielding in a permanent fault during the operational phase [1].

Electro migration is a mechanism affecting interconnects. Electrons flowing through the wire, collide with metal atoms, and may force them to migrate [107]. By this, voids in the metal wire occur that increase the resistance of the wire or even completely disconnect it. The migrated material is settled in other sites and may cause there a short. This effect is increased by further shrinking the feature size, because current density increases [49]. Multiple vias can be used as backup for connecting wires [1]. In some special cases the effect of EM is reversible [5], but this solution cannot be applied universally, because it requires an inversion of the current flow. Very similar to EM is stress migration, where metal atoms migrate due to mechanical stress caused by thermal expansion of the material [175]. Finally it should be noted that models for NBTI, HCI, and EM contain the temperature as an important factor [112, 177]. Thus, the presented aging effects strongly depend on the temperature and stress profile of the system, and the temperature and stress profile depends on the executed user application. Therefore, controlling the stress profile of the application will help to control the temperature profile, which finally can delay the aging effects [184].

1.1.2 Means

The purpose of the means in figure 1-2 is the improvement of our confidence in the dependable system. The classification of faults as shown in figure 1-3 helps to develop means that can rely on particular properties of faults belonging to certain fault classes. The means *fault avoidance*, *error removal*, and *fault tolerance* are related in figure 1-5 with the faults introduced in particular stages of the life-cycle of a processor-based system. The life cycle starts with the *development phase*, which is further divided into the stages shown as arrows in the upper part of figure 1-5. The impairments during these stages of the development phase come from *specification faults*, *design faults*, *implementation faults* and *manufacturing faults*. For example, implementation faults may be introduced into the source code of a software- or hardware-component, when programming languages or hardware description languages are used. Manufacturing faults will occur in the hardware components due the reasons already described in section 1.1.1.5, and they will occur in software components probably due to flaws in the used compiler.

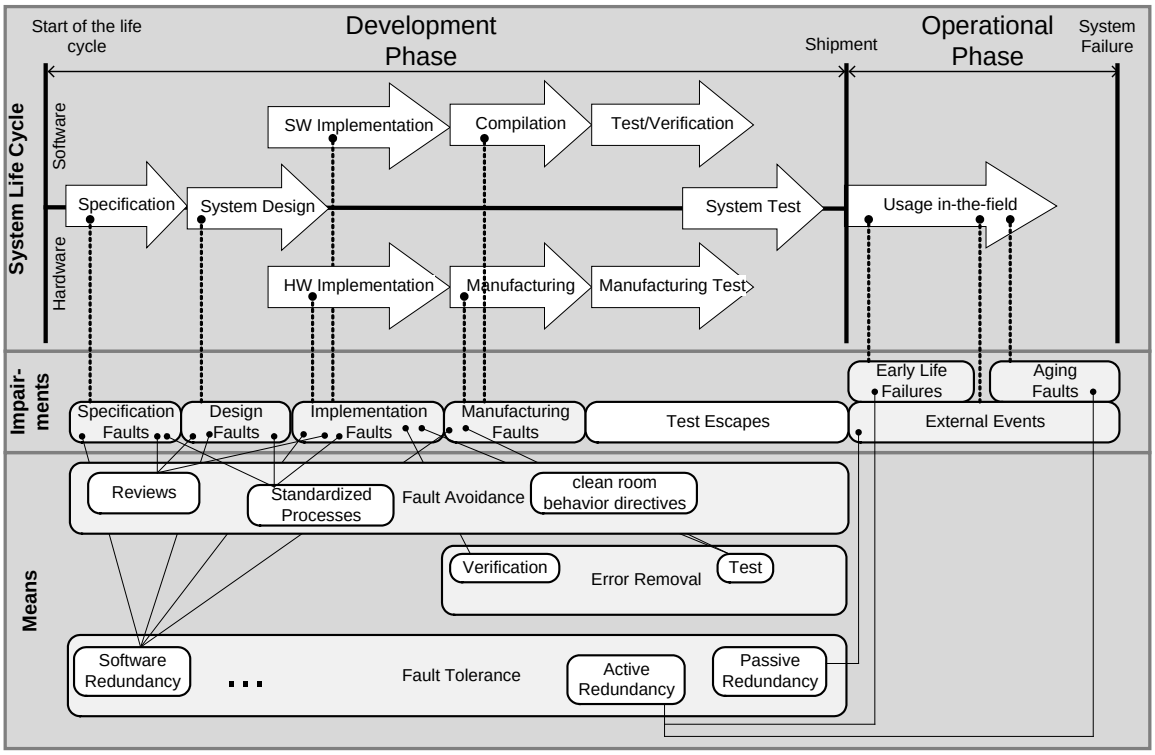


Figure 1-5: Life-cycle of a system related with impairments and means. Impairments are related by dotted lines with life-cycle phases. Means against impairments are related with them by solid lines.

1.1.2.1 Fault Avoidance and Error Removal

Fault avoidance techniques are employed during the development phase, in order to avoid the introduction of development faults into the system (see figure 1-5). These techniques include, for example, review processes, standardized specification- and design processes, and many other quality control methods [140]. However, at least it is the objective of fault avoidance techniques to keep the number of latent development faults in the system as low as possible. In larger systems it is very likely that some development faults cannot be avoided, no matter how carefully fault avoidance techniques are applied. Therefore, error removal techniques are used as another mean for minimizing the presence of latent faults in the manufactured device by removing detected faults. For example, the verification and test of software components, as well as the manufacturing test of dies are used for error detection. Detected implementation faults in the software components can be fixed. Detected manufacturing faults in hardware components not. Either the defect die is rejected or it is sold with degraded functionality. For memory units the fabrication with redundancy is state of the art, such that faulty bit-lines can be replaced with redundant backup elements [106, 126].

1.1.2.2 Fault Tolerance

No matter how much effort is spend during the development phase on fault avoidance and error removal, there is a very high probability that faults remain

undiscovered until the system is in operation (in software and hardware components). For example, there are dozens of design faults in AMD and Intel cores [4]. Moreover, permanent operational faults will be introduced into the system during the operational phase by aging and single event effects already described in section 1.1.1.5. When these dormant faults and the operational permanent faults become active during the operational phase, then an error occurs in the system. Means for handling these errors are called fault tolerance techniques. They are incorporated into the system during the development phase, but they become active during operational phase (see figure 1-5). In general, fault tolerance is defined as follows:

Definition 1-7: (Fault Tolerance [140])
Fault tolerance is the ability of a system to continue to perform its task after the occurrence of faults. The ultimate goal of fault tolerance is to prevent system failures from occurring.

The first practical fault tolerance techniques have been developed in the late 1940's and early 1950's when the computers were built from very unreliable relay and tubes [15]. Since then it is well accepted that fault tolerance is achieved by *redundancy*. In [94] it is stated:

> *"All of fault tolerance is an exercise in exploiting and managing redundancy".*

Thus, the broad variety of fault tolerance techniques is classified according to the managed redundancy type.

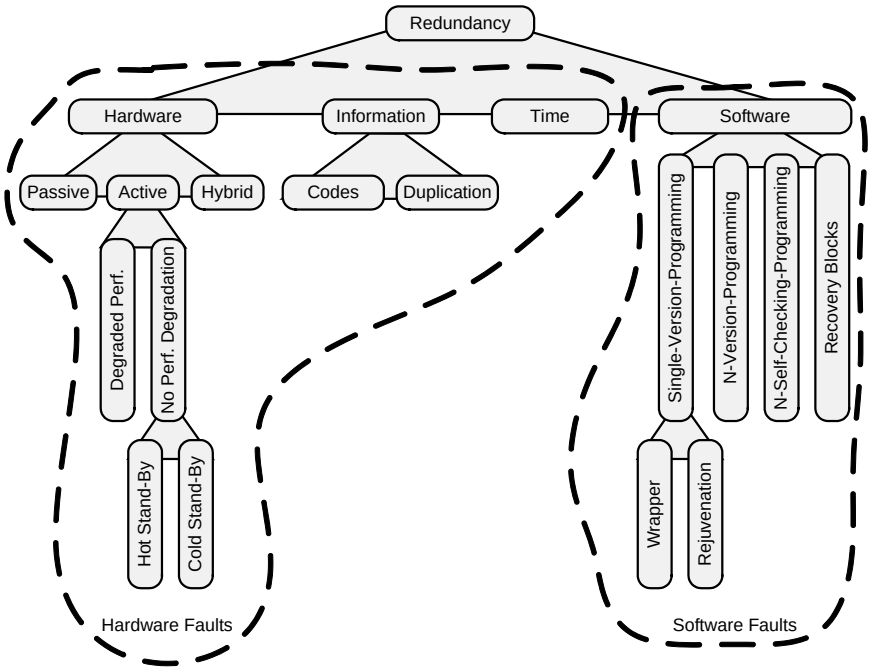


Figure 1-6: Classification of redundancy.

This classification is shown figure 1-6 and embraces hardware redundancy, information redundancy, time redundancy and software redundancy. Software redundancy is typically used for detecting and handling specification, design, implementation, and manufacturing faults in software components. Faults affecting the hardware components of a system are handled either by information, time, or hardware redundancy.

Software redundancy

The basic concept of *software redundancy*, except for single-version programming, is diversity. I.e., multiple programs that compute the same function are developed independently from each other[4]. The system runs several of these programs and compares the outputs. Existing software-redundancy approaches vary in the way how these diverse programs are developed, activated, and when their results are compared. In *N*-version programming [40] *N* different versions of the program are implemented and run in parallel. Their outputs are compared and the result is selected by a majority vote. *N* different versions of a program are also available in *N*-self-checking programming [104]. Each of them is capable of performing an acceptance test. The voter only votes on accepted outputs. In the recovery-block-approach [143] *N* different versions of a program and a single centralized acceptance test are available. A program is only invoked, if the acceptance test of the previously invoked programs has failed. I.e., the result of the first program whose output is accepted is used.

Information Redundancy

Information redundancy is based on redundancy in data. Usually it is applied for detecting errors that occur during transmitting or storing data. In its simplest form, data is just replicated in the system. However, the most widely used information redundancy technique is *error control coding* [64]. In general error control coding is the mapping of the original data word into a code word by adding redundancy. Therefore, the binary code word has usually more bits than the original binary data word. The *code* is the set of all code words. After transmitting or storing a code word, the original data word is reconstructed from the received word by a decoding process. When this decoding process can detect and correct errors in the received word, then the code is called an *error correction code* (*ECC*). When errors can be detected, but not corrected, then the code is called an *error detection code* (*EDC*). In general, the error detection is based on

[4] It is interesting to note that this concept of diversity has been already described in the context of Babbage's Calculating Engine (see the quotation on page 5).

checking whether the received word is a code word or not. If the received word is a code word, then it is assumed that no error has been occurred. Therefore, errors that corrupt a code word in such a way that another code word originates cannot be detected.

The usage of ECCs and EDCs for protecting memories and processor registers against transient faults is state of the art in current designs of server processors [172] or embedded processors for avionic [66]. However, these techniques cannot be applied in a straight forward manner as fault tolerance techniques for arbitrary combinatorial logic that performs some kind of operation $f(x_1,\ldots,x_n)$ on the input operands $x_1,\ldots,x_n$, because the operation usually changes the data words $x_1,\ldots,x_n$. Special codes will be employed for this purpose, where the encoding can be considered as a homomorphism h, and a simple checker-function f^h exists, such that

$$h(f(x_1,\ldots,x_n)) = f^h(h(x_1),\ldots h(x_n)). \tag{1-1}$$

I.e., concurrently to the computation of the result by function f, a reference value is computed by the checker-function f^h based on the encoded operands [13]. Whether or not f has performed correctly, is checked by encoding the result of f with the homomorphism h and comparing this value with the result of the checker function. As an example consider an arithmetic code for an adder [94] with $f(x_1, x_2) = f^h(x_1, x_2) := x_1 + x_2$, and $h(y) := 3 \cdot y$. The Berger code has been also used as error detection code for arithmetic logic units (ALU) [110] and floating point arithmetic units [137]. However, the draw-back of such an approach is that each operation f in the ALU requires a corresponding operation f^h that should be much simpler to implement in hardware. Unfortunately this is not the case for the Berger code and the arithmetic code. In [137] overheads of more than 100% are reported for the checker-unit, when the Berger code is used. Also the presented arithmetic code needs a complete adder in the checker-unit. Therefore, the checker-unit has at least the same size as the original unit for f. This means that a simple duplication of the original unit will be more beneficial in these cases, because the duplicated unit can perform exactly the same operation as the original unit, and therefore additionally serve as backup-unit. However, in the next section it is shown that the presented type of encoding may be useful in combination with time redundancy.

Time Redundancy

The fundamental concept of *time redundancy* is to perform the same computation multiple times by using the same hardware components. The results obtained by multiple computations are either compared with each other or will undergo an acceptance test. This is conceptually very similar to software redundancy, when the hardware component is a processor that executes a piece of software. The

difference to software redundancy is that in each run the same piece of software is executed. Thus, time redundancy cannot be used for detecting software faults. The intention of time redundancy is the detection of temporary faults. The detection of permanent faults is not possible by using the simple concept of re-execution, because the computations are performed on the same piece of hardware. However, detecting permanent faults is possible by the combination of time redundancy with encoding. For encoding a homomorphism h, as presented in the previous section about time redundancy, is used. Because the computations of f and f^h (see equation (1-1)) must be executed by the same component, the homomorphism h must be selected in such a way that

$$h(f(x_1,\ldots,x_n)) = f(h(x_1),\ldots,h(x_n))$$

holds. RESO (re-computation with shifted operands) is a popular approach that uses this technique [133, 134]. There, shift operations are used as encoding function. Another form of encoding is the bit-wise inversion of operands [140]. I.e., for function f must hold

$$\overline{f(x_1,\ldots,x_n)} = f(\overline{x_1},\ldots,\overline{x_n}).$$

Such a function f is called self-dual [140]. The advantage of using time redundancy for detecting permanent faults is the little hardware overhead needed for the implementation. However, this comes at the cost of at least a doubled execution time. Moreover, the concept cannot be used in a universal fashion, because for each function f that should be checked with time redundancy for permanent faults a dedicated encoding function h is needed.

Hardware Redundancy

Hardware redundancy employs redundant hardware components. According to figure 1-6 it is further divided into active and passive redundancy. *Passive hardware redundancy* does not perform *fault isolation*; i.e., a defect component is not taken out of operation and a fixed configuration of $N \in \mathbb{N}_+$ redundant components in the system is maintained. Computations are performed in parallel on all N components. The components receive the same inputs and their results are compared with each other. In a *double modular redundancy* (*DMR*) system ($N = 2$), as shown in figure 1-7 (a), the comparison allows for error detection only. For $N > 2$ a majority vote can be performed, and the result that is delivered by the majority of the components is selected as output of the voter. Figure 1-7 (b) shows the well known *triple modular redundancy* (*TMR*) system ($N = 3$), which has been already proposed in the 1950's by von Neumann in order to built more reliable relay networks [186].

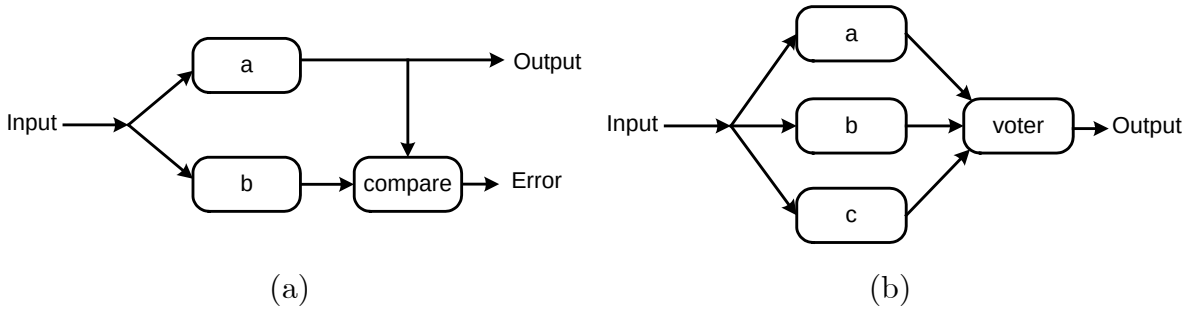


Figure 1-7: (a) DMR system with redundant components *a* and *b*. (b) TMR system with redundant components *a*, *b* and *c*.

Passive redundancy is typically used for handling temporary faults, because the static configuration of the system allows for fast error detection and error handling. On the other hand, components containing a permanent fault are not isolated. Thus, an erroneous result of such a component is only masked by the voter. This is sufficient for handling temporary faults, because it is very likely that the temporary fault will disappear, before another temporary fault appears. Therefore, a method to handle a single error at any point in time is sufficient. However, in nano-scaled systems it becomes likely that permanent faults accumulate over time and appear together with a temporary fault, leading to a situation where faults are present in multiple components of the system. Handling of n faults in n different components requires at least $n + 2$ redundant components in the system, assuming that all faulty components will produce pairwise different results. At least $2n + 1$ redundant components are needed without this assumption. In both cases the handling of multiple faults will create a strong hardware and power consumption overhead, which is not acceptable for many embedded systems.

Active hardware redundancy is typically used to overcome this overhead problem by *fault isolation* when handling multiple permanent faults. If active hardware redundancy is used in a system, then the configuration of the system is changed, depending on its current fault state, such that permanently faulty components are isolated; i.e., they are taken out of operation. This fault isolation requires a more complex administrative organization, including the error detection, the localization of the component containing the fault, and the *reconfiguration* of the system, such that the defect component is taken out of operation. These administrative steps can be performed either when the system is on-line or when the system is off-line. In *on-line* mode, the system delivers its normal service (e.g., the processor executes the user application). For this reason *concurrent error detection* can be used for error detection. By concurrent error detection transient faults and permanent faults that appear during the execution of the user application can be detected. Passive hardware redundancy, information redundancy, or time redundancy may be used for this purpose. Thereby the concurrent error detection may be also used

for fault localization. For example, in sub-system level TMR [94], the sub-systems of a system are tripled. Each tripled sub-system receives three identical inputs and produces three outputs by using three voters. The three outputs are used as inputs of other tripled sub-systems. Implementing this redundancy scheme for a large number of very small sub-systems allows for fine-grained on-line fault localization at sub-system level, but at the expense of a large overhead due to the large number of voters and redundant components.

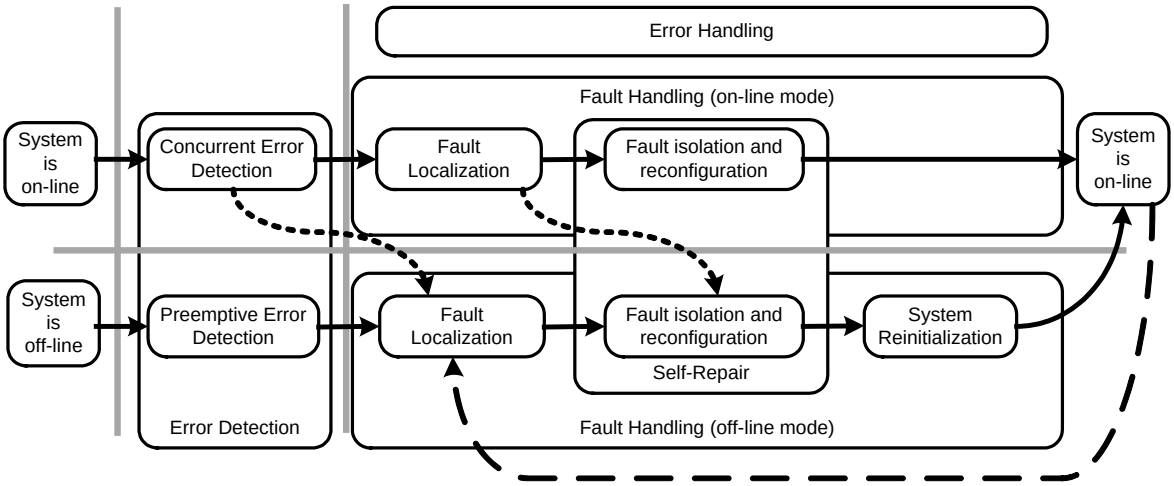


Figure 1-8: Implementation of fault tolerance when active hardware redundancy is used.

Fault handling is the process of fault localization and reconfiguration. In on-line mode, *fault handling* is performed concurrently with *error handling*. Thereby, error handling is the process of masking errors in the running application until fault handling is completed (see upper part of figure 1-8). The combination of passive and active redundancy techniques for fault and error handling in on-line mode yields *hybrid hardware redundancy* schemes. Examples of such systems are NMR-with-spares, triple-duplex, or pair-and-spare systems [94, 140]. In all of these systems there is enough redundancy available such that a defect component can be replaced without interrupting the normal service of the system. This is, of course, at the expense of hardware overhead, which is for example, for triple-duplex more than six times of the original size of the system.

In the *off-line* mode the system is powered on, but it does not perform the normal services. For example, the processor is in idle-mode or the user application has been not started after power on. Then, most of the resources of the system can be used for *preemptive error detection* and fault handling (see lower part of figure 1-8). Preemptive error detection is used for the detection and localization of permanent faults in off-line mode. Built-in self-test (BIST) or software-based self-test (SBST) may be used for this purpose. Compared with on-line test methods, more time for test and fault localization is available in the off-line mode, because the system does not have to deliver its service continuously. The development of

such a diagnostic test is the topic of chapter 5. Furthermore, some parts of the error detection and fault handling may be done in on-line mode, but then the system changes into the off-line mode (shown by the dotted arcs). For example, concurrent error detection may be used for detecting temporary and permanent faults during the runtime of the application. After detecting an error, the system switches into off-line mode for performing a fine-grained but time consuming reconfiguration. Thereby the fault localization delivers information about the defective component to the method used for reconfiguration and fault isolation. It is sufficient to locate a defective component at that granularity level that is used by the subsequent reconfiguration step. For example, if the subsequent reconfiguration step only allows the replacing of complete processors in a multi-processor system, then it is sufficient to locate the defective processor. Which particular component inside of the processor is faulty is not of interest for the reconfiguration. There are two basic strategies for replacing a faulty component during reconfiguration. Either

- the functionality is allocated to *spare components* that were not in use before, or
- the functionality is allocated to components that were already in use before.

In the latter case, the system is usually affected by performance degradations, because other components that were already in use must overtake some functionality of the faulty component. As a consequence, the system needs more time for providing its service or the accuracy of the results is reduced [135, 159]. In the former case, the system performance is not degraded (see also the classification in figure 1-6). Sometimes this is also referred to as *self-repair*, while the latter case is named *graceful (performance) degradation*. In this thesis the term self-repair is used for the fault isolation and reconfiguration process, no matter whether the performance is affected by the reconfiguration or not. Distinguishing between reconfiguration methods with and without performance degradation is not always reasonable, because the same reconfiguration method may cause for some faults a performance degradation and for other faults not. The software-based self-repair method presented in chapter 3 is such a method. There, different faults may cause different reconfigurations of the system. Thereby, one of these reconfigurations will cause a performance degradation and another reconfiguration will not.

When the reconfiguration step replaces the defective component by a spare component, then this spare component may be in a cold standby or in a hot-standby mode. *Cold standby* means that backup components are not powered. In *hot standby*, backup components are powered. Usually, replacing a faulty component with a hot standby component is accomplished within a shorter time than replacing it with a cold standby component [140]. On the other hand, hot

standby components may suffer from degrading effects like aging that do not affect cold standby components, and they need extra power.

When the reconfiguration is done in off-line mode, then a final recovery step is needed in order to bring the system back into an operational mode. This recovery may be a simple restart of the system. But it can be also a more complex roll-back of the system state, up to a previous faultless state taken at a checkpoint. When the reconfiguration is done in on-line mode, then the system is already in operational mode. For this reason a recovery is usually not needed. However, in this case the system may switch into off-line mode after some time, in order to perform a more sophisticated reconfiguration (shown in figure 1-8 by the backward arrow). By the more sophisticated reconfiguration the system is adapted in an optimized way to the current fault state, such that the required functionality is provided in the best way, given the current fault state. The motivation for such an adaptation comes from the on-line fault handling step that will bring back the system as quick as possible into an operational state. For this reason the reconfiguration is often performed in a coarser grained manner. For example, a complete processor is taken out of operation during on-line reconfiguration. For the sophisticated reconfiguration in off-line mode more time is available for a diagnosis of the defect processor. If possible, the reconfiguration is performed for the sub-components of the processor such that the processor can be used again for executing particular user applications. This allows for a better utilization of all functioning components in the system. On the other hand, the more sophisticated off-line diagnosis and reconfiguration require more time and computational resources than available in the on-line reconfiguration step.

1.1.2.3 An Orthogonal Classification of Fault Tolerance Techniques

The typical classification of fault tolerance methods shown in figure 1-6 refers to the type of redundancy. However, the redundancy in a system must be managed somehow. For this reason an orthogonal classification of fault tolerance methods, which is shown in the vertical direction of figure 1-9, refers to the management of redundancy.

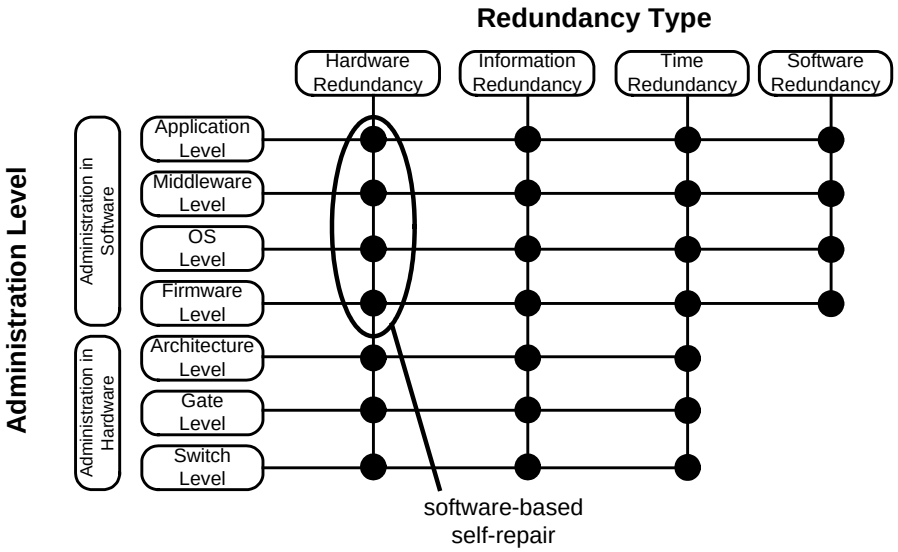


Figure 1-9: Classification of fault tolerance methods according to the kind of administration.

The management can be done at different system layers. Thereby, hardware based management methods are used for administrating redundancy at switch level, gate level, or architecture level. Software-based methods are used for the management at firmware level, operating system level, middleware level, or application level. For example, the hardware redundancy in a TMR system can be managed either by hardware – then the voter is implemented as a hardware structure – or by software; then the voter is implemented as a piece of software that compares the results of three hardware components. In the latter case software is used for administrating hardware redundancy, and in the former case hardware is used for administrating hardware redundancy. In the same manner the encoder and decoder for the administration of information redundancy can be implemented either in software or in hardware. Time redundancy can be also administrated in software, e.g. by creating checkpoints and repeating the execution of a piece of software, but also by a re-execution of instructions that is controlled by hardware [41]. The software-based self-repair presented in chapter 3 uses software for the administration of hardware redundancy.

Cross-layer approaches were proposed recently that employ the managing of redundancy at various levels [48, 50, 79]. This can be very helpful for handling faults at a level where they can be handled best with respect to time- and power-constraints [79]. Moreover, faults can be handled with various granularities. I.e., if the management at a particular level cannot handle the occurred fault(s), then the management at another level may perform the reconfiguration. The proposed software-based self-repair approach in chapter 3 can be seen as part of such a cross-layer approach. The hybrid approaches in chapter 4 provide some applications of coupling hardware-based administration of redundancy with software-based administration at different system layers.

1.1.3 Attributes

The attributes shown in the dependability-tree in figure 1-2 allow for the quantification of the negative impact of faults, errors, and failures and the positive impact of fault prevention, fault removal, and fault tolerance on the dependability. This quantification is part of the *fault forecasting* by evaluating probabilities for the occurrence of failures under given assumptions about the occurrence of faults [15]. The most relevant attribute for the quantification of the impact of fault tolerance methods is the reliability. Strongly connected with the reliability is the mean time to failure.

1.1.3.1 Reliability

An improvement of the reliability increases the probability that a systems performs correctly within a specified period of time. This attribute may be used for characterizing the impact of fault tolerance methods on systems with a specified mission time.

Definition 1-8 (Reliability, taken from [140]):
The *reliability* of a system is a function $R(t)$ of time $t \in \mathbb{R}$, defined as the conditional probability that the system performs correctly throughout the interval of time $[t_0,t]$, given that the system was performing correctly at time t_0.

The reliability can be determined empirically by considering a sufficiently large number N of identical systems that are functioning correctly at time t_0. All of these systems are taken into operation at time t_0. Sometime later, at time $t \geq t_0$, the number $F(t)$ of failed systems and the number $C(t)$ of correct functioning systems is determined. Now the reliability $R(t)$ is simply

$$R(t) = \frac{C(t)}{N} = \frac{N - F(t)}{N}. \tag{1-2}$$

It is important to notice that the reliability is defined only for systems that run correctly throughout the time interval $[t_0,t]$. I.e., a system that has failed within this time interval is considered to be faulty forever. Nevertheless, the reliability definition applies to fault-tolerant systems that support self-repair functionality. The important fact of the given reliability-definition is how the term "performs correctly" is defined. At a first glance it means that the system does not produce a failure. I.e., an error inside the system is allowed to occur, as long as this error is not propagated to the outside of the system. This, for example, applies to systems employing fault masking. However, even in systems that allow failures to occur, the reliability-definition applies, if this conforms to the specification of the system and the system is able to recover autonomously to operational state. I.e., a faulty output may be accepted, but the system must be able to return to normal

operation after (off-line) fault handling; probably by a restart. The system does no longer perform correctly, when the self-repair capability is exhausted, and an autonomous fault handling is not feasible anymore.

In this situation a repair function may be applied to bring the system back into an operational mode. Please note that there is a fundamental difference between *self-repair* and *repair*. The self-repair is accomplished autonomously by the system. A repair process is performed by an external entity that removes a faulty component physically from the system and replaces it by another one. For systems in which a failure is handled by a repair process, the attribute reliability is no longer meaningful, because then the system was not performing correctly all the time. Such systems are typically characterized by the attribute *availability*. The availability is a function of time, $A(t)$, defined as the probability that a system is operating correctly and is available to perform its functions at time t [140]. The availability differs from the reliability in that way that a particular point in time, t, is considered, instead of a time period. This attribute applies for example to servers that provide some internet service.

1.1.3.2 Mean Time to Failure

The *mean time to failure* (*MTTF*) expresses the expected time for a single system until it stops providing the correct service. In order to determine the MTTF empirically, consider a sufficiently large number N of systems placed into operation at time $t_0 = 0$, and all of them performing correctly at time $t_0 = 0$. Let t_i be the first point in time when system i does not perform correctly, whereby $i \in \mathbb{N}$ - $\{0\}$. Then the total runtime TRT_N of all the N systems together is

$$TRT_N := \sum_{i=1}^{N} t_i \,.$$

The mean time to failure is now obtained by

$$MTTF_N := \frac{TRT_N}{N} = \sum_{i=1}^{N} \frac{t_i}{N} \,. \tag{1-3}$$

In order to obtain the relationship between reliability and mean time to failure, suppose that the variables t_i are given in ascending order. That is: $t_i < t_{i+1}$ for all $1 \leq i < N$. Now recall that t_i is the first time that system i has failed. Thus, the number of correct functioning systems in the time interval $[t_i, t_{i+1})$ is given by $C(t_i)$ as it was defined in section 1.1.3.1. Therefore, the total time that all functioning systems are running within this interval is $(t_{i+1} - t_i) \cdot C(t_i)$. The total runtime TRT_N that all the systems are running together can be also defined by summing up the total runtimes within all the time intervals $[t_1, t_2),\ldots, [t_{N-1}, t_N)$:

$$TRT_N \coloneqq \sum_{i=0}^{N-1} (t_{i+1} - t_i) \cdot C(t_i)\,.$$

Now the definition of the reliability from formula (1-2) is used to replace $C(t_i)$:

$$TRT_N \coloneqq \sum_{i=0}^{N-1} (t_{i+1} - t_i) \cdot R(t_i) \cdot N = N \cdot \sum_{i=0}^{N-1} (t_{i+1} - t_i) \cdot R(t_i)\,.$$

Using the definition of $MTTF_N$ in equation (1-3) yields:

$$MTTF_N \coloneqq \sum_{i=0}^{N-1} (t_{i+1} - t_i) \cdot R(t_i)\,.$$

For very small time intervals $(t_{i+1} - t_i)$ this becomes approximately the area underneath a continuous reliability function $R(t)$ within the interval $[0,t_N]$, and for $N \to \infty$ the MTTF becomes

$$MTTF \coloneqq \int_0^\infty R(t)\, dt\,. \tag{1-4}$$

From equation (1-4) follows that reliability and mean time to failure will not necessarily have a simple correlation. As an example consider a system A and a system B that is a TMR system composed of three identical components A. The reliability function of A is given by $R_A(t)$ and the reliability function of B by $R_B(t)$:

$$R_A(t) = e^{-0.1 \cdot t} \qquad\qquad R_B(t) = 3e^{-2 \cdot 0.1 \cdot t} - 2e^{-3 \cdot 0.1 \cdot t}$$

Both functions are plotted in figure 1-10. It can be noticed that for $t < 7$ system B has a better reliability than system A. I.e., the probability that system B has a failure within the time interval [0,7] is lower than for system A. For $t > 7$ this situation changes. Then it is more likely for System B to have a failure than for system A. According to equation (1-4) the mean times to failure (MTTF) for both systems are given by:

$$MTTF_A = \int_0^\infty e^{-0.1 \cdot t}\, dt = 10 \qquad MTTF_B = \int_0^\infty \left(3e^{-2 \cdot 0.1 \cdot t} - 2e^{-3 \cdot 0.1 \cdot t}\right) dt \approx 8,3$$

System B has a shorter MTTF than system A. Thus, one may prefer system A prior system B, when it is the goal to have a system with a long life time. However, when the mission time for the system is known in advance to be less than 7, and the probability that the systems fails within its mission time should be as low as possible, then system B is favorable.

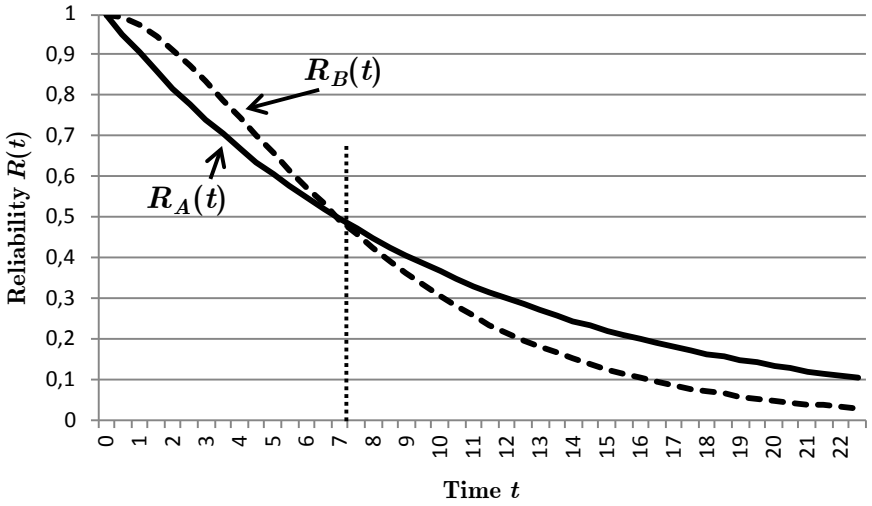


Figure 1-10: Reliability plot for system *A* and system *B*.

The reason for the lower MTTF of B, although B contains more redundancy, is that the computation of the MTTF takes the full life time ($t \to \infty$ in equation (1-4)) of all systems into account. In figure 1-10 this fact can be noticed for $t > 7$. In the beginning, for $t < 7$, more systems of type B survive. When computing the MTTF only for the time interval [0,7], then system B will also have a better mean time to failure than system A. But for $t > 7$ more systems of type A than systems of type B will survive. By this the MTTF of system A will outperform the MTTF of system B. But from a practical point of view, the time period for $t > 7$ is of very little interest, because the probability that a system has survived up to this point in time is only 50%. Moreover, a mission time T that is of practical relevance would be much lower than 7 in this example, such that the probability that a system has survived until time T is much higher than 50%. For example, the reliability requirement for highly reliable flight control systems is 0.999999 for $T = 5$ hours [146].

Thus, when a fixed mission time T is known, then it is expected that the fault tolerant system has a higher reliability than the original system during the full mission time. This can be expressed, for example, by the *reliability improvement factor* (*RIF*) that is defined as:

$$RIF = \frac{1 - R_A(T)}{1 - R_B(T)},$$

where R_A is the reliability of the non-fault tolerant system and R_B is the reliability of the fault tolerant system [100]. The RIF should be larger than one for a reliability improvement. When the mission time is not known, then the *mission time improvement factor* (*MTIF*) may be used for evaluating the benefit of the used fault tolerance method [100]:

$$MTIF = \frac{T_B}{T_A}.$$

Thereby T_B is the time, when system B reaches a prior specified reliability R; i.e. $R = R_B(T_B)$, and T_A is the time when system A reaches the specified reliability value R; i.e. $R = R_A(T_A)$.

1.1.3.3 Reliability Modeling

The reliability was introduced in equation (1-2) based on empirical observations given by N, $C(t)$, and $F(t)$, where $F(t)$ is the number of non-operational systems at time t, $C(t)$ is the number of operational systems at time t, and $N = F(t) + C(t)$ is the total number of systems at any time $t \geq 0$. This equation can be further converted into

$$R(t) = \frac{C(t)}{N} = \frac{N - F(t)}{N} = \frac{N}{N} - \frac{F(t)}{N} = 1 - \frac{F(t)}{N}. \tag{1-5}$$

In order to perform a fault forecasting without an empirical observation, a model is needed for forecasting $F(t)$ and $C(t)$. This forecasting usually employs the *failure rate*, which is the rate of failing systems at a particular time t. Thereby, the relation between the failure rate, $F(t)$, and $C(t)$ is obtained by the differentiation of the continuous function $F(t)$:

$$\frac{dF(t)}{dt} = \lim_{h \to 0} \frac{F(t+h) - F(t)}{h}. \tag{1-6}$$

$F(t+h) - F(t)$ is the number of failed systems within time interval h. For $h \to 0$ this is approximately the instantaneous number of failing systems at time t. When dividing this number by the number of systems that are operational at time t, which is $C(t)$, the rate (or fraction) of failed systems is obtained, which is well known as failure rate $z(t)$:

$$z(t) = \frac{1}{C(t)} \cdot \frac{dF(t)}{dt}. \tag{1-7}$$

If $z(t)$ is a constant function over time t, then it is commonly denoted as λ. Now the relationship between reliability and failure rate is considered as it can be found in a similar manner in many textbooks, e.g. [94, 116, 140]. The deviation of $F(t)$, which is also part of the failure rate function (see equation (1-7)), is related with $R(t)$ by computing the deviation of both sides of equation (1-5)

$$\frac{dR(t)}{dt} = \frac{d\left(1 - \frac{F(t)}{N}\right)}{dt} = -\frac{1}{N} \cdot \frac{dF(t)}{dt}. \tag{1-8}$$

This formula can be also written as

$$-N \cdot \frac{dR(t)}{dt} = \frac{dF(t)}{dt} \tag{1-9}$$

Replacing in equation (1-9) the deviation of $F(t)$ with equation (1-7) yields the relation between the failure rate and the deviation of the reliability:

$$z(t) = -\frac{N}{C(t)} \cdot \frac{dR(t)}{dt}. \tag{1-10}$$

From equation (1-5) it is known that

$$\frac{1}{R(t)} = \frac{N}{C(t)}, \tag{1-11}$$

Replacing $N/C(t)$ in equation (1-10) with $1/R(t)$ yields:

$$z(t) = -\frac{1}{R(t)} \cdot \frac{dR(t)}{d(t)}. \tag{1-12}$$

This can be written as the differential equation:

$$-R(t) \cdot z(t) = \frac{dR(t)}{d(t)}. \tag{1-13}$$

The general solution of this equation with $R(0) = 1$ is given by

$$R(t) = e^{-\int z(t)dt}. \tag{1-14}$$

Equation (1-14) describes the general relation between reliability and failure rate. If $z(t)$ is assumed to be a constant λ, then the relationship between failure rate and reliability simplifies to

$$R(t) = e^{-\lambda \cdot t}.$$

Furthermore, according to equation (1-4) the mean time to failure becomes

$$MTTF = \frac{1}{\lambda}. \tag{1-15}$$

An increasing or decreasing failure rate $z(t)$ is often modeled by using the Weibull-distribution [94], which yields the failure rate function:

$$z(t) = \lambda \cdot \beta \cdot t^{\beta - 1} \tag{1-16}$$

For $\beta = 1$ this failure rate function simplifies to the constant failure rate λ. For $\beta < 1$ $z(t)$ becomes a decreasing function over time, and for $\beta > 1$ it becomes an increasing function over time. The corresponding function for the reliability is

$$R(t) = e^{-\lambda \cdot t^{\beta}}.$$

In order to perform a fault forecasting using such a probabilistic model, the assumptions about the failure rate must be validated. In industrial practice, the failure rate of semiconductor devices is determined by performing an accelerated operating life test on a large number of randomly selected devices [2]. Acceleration is usually achieved by operating the device at high temperatures and/or higher voltages for several thousand hours, which sums up to several million total device hours[5]. Based on the distribution of the observed faults within this time interval, the assumption about a constant, falling or increasing failure rate of the considered device type can be validated. Moreover, the parameters λ and β can be estimated, based on the total number of devices put into operation, the distribution of failed devices over time, and the simulated operation time. For real systems it is common to observe a behavior of the failure rate as it is shown in the bathtub curve in figure 1-11.

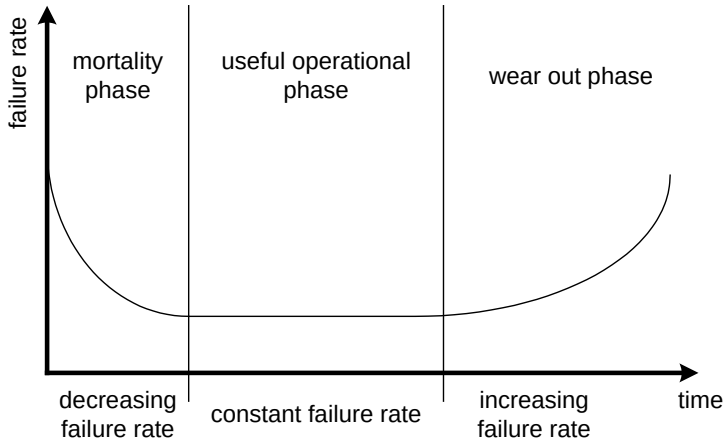


Figure 1-11: Bathtub curve of the failure rate.

Before the useful operational phase starts, the system undergoes an infant mortality phase with a decreasing failure rate. During this phase, manufacturing flaws emerge as permanent faults. This phase is overcome by stressing the devices using a burn-in test that is performed by the device manufacturer. The problem of a burn-in test is that it also stresses devices without manufacturing flaws. For these devices the useful operational phase may be shorten by the burn-in test. The problem of detecting as many faults as possible during burn-in test can be relaxed by including self-repair techniques, as it is claimed for example in the ITRS roadmap [3]. Then, the early-life failures can be handled during the operational phase.

A constant failure rate is typically observed for the useful operational phase, due to random effects like single event latch-up. Aging effects like TDDB, HCI, NBTI and EM will occur more likely during the wear-out-phase. Therefore, the failure

[5] http://rel.intersil.com/docs/rel/calculation_of_semiconductor_failure_rates.pdf

rate increases during this time period. Nevertheless, some of the aging effects will already occur before the wear-out-phase starts, because their occurrence is promoted by manufacturing flaws like traps in the gate oxide and wider parameter distributions as they have been discussed in section 1.1.1.5. A very popular model for estimating the mean time to failure of a system is RAMP [175]. RAMP is based on various practically used industry models for calculating the mean time to failure of a component. For each aging effect a dedicated model exists that is based on numerous technology dependent parameters. RAMP provides a combination of these models, such that all of these effects are taken into account for a single component. The fundamental assumption is that the failure rate is constant [177], although it is clear that the failure rate for aging effects is not constant over timer. In this thesis also a constant failure rate in components is assumed. Moreover, the reliability $R(t)$ instead of the mean time to failure is used for quantifying the benefit of the proposed fault tolerance methods, because the reliability allows for a better analysis over time.

1.1.4 Reliability Estimation for Fault Tolerant Systems

In the previous section, a probabilistic model for the forecasting of the reliability of a component was considered, whereby the reliability function for that component was defined by using the failure rate. Now fault tolerant systems are modeled, which are composed of multiple components, and for each component its reliability is given by a reliability function. I.e., the reliability of a system has to be defined by using the reliability functions of its components. Thereby it is allowed that particular components of the system do not perform correctly, because they are affected by a permanent fault. It is assumed that faults in different components appear independent from each other. In order to model permanent faults correctly, it is also important that non-functioning components remain faulty forever, because the permanent fault will never disappear. Please note that this assumption does not hold for modeling temporary faults, because a component affected by a temporary fault at time t is faulty at time t, and some time later it is working properly again, if the error caused by the temporary fault has disappeared. The considered techniques for modeling the reliability of fault tolerant systems are based either on combinatorial models or state-space models [146].

1.1.4.1 Combinatorial Models

Basically a combinatorial model considers a fault tolerant system as a set S of components $S = \{C_1,\ldots,C_n\}$, for $n \in \mathbb{N} - \{0\}$. Each component can be either faulty or faultless. Therefore, S can be partitioned into a set O of faultless components and a set $S - O$ of faulty components. All possible states of the system are given by the set of all subsets of S, which is $\wp(S)$. For example, the subset $O = S$

represents the system state in which all components are faultless. Furthermore, for each component $c \in S$ its reliability function is denoted by $R_c(t)$. I.e., $R_c(t)$ is the probability that component c is operational at time t, and $1 - R_c(t)$ is the probability that component c is faulty at time t. Therefore, the probability that the system is in a particular state O at time t is given by

$$P_O(t) = \prod_{c \in O} R_c(t) \cdot \prod_{c \in S-O} (1 - R_c(t)) . \tag{1-17}$$

In order to obtain the probability that a system S is operational at time t the function ok is used. The function $ok : \wp(S) \rightarrow \{0,1\}$ determines for each system state whether the system is still operational in this particular state or not. Now the reliability of S is given by:

$$R_S(t) = \sum_{O \in \wp(S) \text{ and } ok(O)=1} P_O(t) . \tag{1-18}$$

As an example the TMR system in figure 1-7 (b) with a voter is considered. The system is given by the components $S = \{a,b,c,v\}$, where a, b, and c are three identical components and v is the voter. A TMR system is operational as long as the voter and at least two out of the three identical components are operational. Thus, the function ok is defined as:

$$ok(X) := \begin{cases} 1, \text{ if } X = \{a,b,c,v\} \vee X = \{a,b,v\} \vee X = \{a,c,v\} \vee X = \{b,c,v\} \\ 0, \text{ otherwise} \end{cases} .$$

According to equation (1-18) the probabilities $P_{\{a,b,c,v\}}$, $P_{\{a,b,v\}}$, $P_{\{a,c,v\}}$, and $P_{\{b,c,v\}}$ must be accumulated. Thereby $P_{\{a,b,c,v\}}$ is the probability that all components in S are operational, $P_{\{a,b,v\}}$ is the probability that components a, b, and v are operational and so on. Because a, b and c are identical components, it holds: $R_a(t) = R_b(t) = R_c(t)$, and in the following $R(t)$ is used for $R_a(t)$, $R_b(t)$, and $R_c(t)$. The result of equation (1-18) is the well known formula for TMR systems as it can be found in many text books:

$$\begin{aligned} & R(t) \cdot R(t) \cdot R(t) \cdot R_v(t) \\ + & R(t) \cdot R(t) \cdot (1 - R(t)) \cdot R_v(t) \\ + & R(t) \cdot (1 - R(t)) \cdot R(t) \cdot R_v(t) \\ + & (1 - R(t)) \cdot R(t) \cdot R(t) \cdot R_v(t) \\ = & R_v(t) \cdot \left(R^3(t) + 3 \cdot R^2(t) \cdot (1 - R(t)) \right) \end{aligned} \tag{1-19}$$

Please note that this formula applies only to TMR-systems that are affected by permanent faults or by temporary faults that cause a permanent error. If the TMR approach is used for handling temporary faults only, and it is assumed that the components are never affected by permanent faults and the temporary fault disappears together with the caused error before the next temporary fault occurs,

then the reliability of the system is equal to the reliability $R_v(t)$ of the voter for the following reason: When a single temporary fault affects one of the components a, b, or c, then the system is operational, because the voter is faultless. Moreover, such a fault will disappear together with the caused error before another temporary fault appears[6]. I.e., the affected component is operational again, such that the TMR-system remains operational as long as the next temporary fault only affects one of the components a, b, and c again. Only temporary faults that affect the voter will cause a failure of the TMR-system.

The presented combinatorial approach is very general, but it requires the consideration of all possible system states. For a larger number of components this may becomes infeasible. Therefore, various approaches exist in order to simplify the representation of the system states. Among them are M-of-N systems, reliability block diagrams, fault trees, and reliability graphs. Some of them are briefly reviewed in the next sections.

M-of-N Systems

An *M-of-N system* is composed of N identical components. At least M of them must operate properly for a functioning M-of-N system. For example, the TMR-system (without the voter) is a 2-of-3 system. The reliability of such a system is computed by

$$R_{M,N}(t) = \sum_{i=M}^{N} \binom{N}{i} \cdot R^i(t) \cdot (1 - R(t))^{N-i} . \tag{1-20}$$

Each summand in this formula represents the probability that exactly i out of N components are functioning. Thereby, the binomial coefficient

$$\binom{N}{i} = \frac{N!}{(N-i)! \cdot i!}$$

is the number of system states with exactly i functioning components. All of these system states will have the same reliability, because all components in the system are identical. Thus, it does not matter which one of these components are the i functioning components.

Reliability Block Diagram

Traditionally, a *reliability block diagram* (*RBD*) is represented by an undirected graph with a source and a sink. Some examples are shown in figure 1-12. The nodes represent the components of the system. Edges represent dependencies between components regarding their operational dependencies. They do not

[6] This can be considered as a repair of the affected component.

represent physical connections between the components. The system is operational as long as there is a path from the source (typically the edge on the left side) to the sink (typically the edge on the right side) that contains only correct functioning components. A simple way for modeling a system with a RBD is the usage of serial and parallel composition of components, as shown in figure 1-12 (a) and (b). Thus, a reliability block diagram is obtained in an inductive manner, by consecutively applying both types of composition rules, starting with the elementary components of the system. Thereby each elementary component is used only once in the RBD.

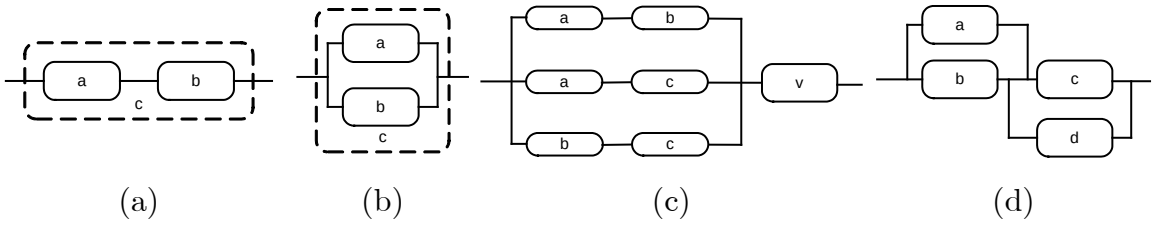


Figure 1-12: (a) Serial composition of two components. (b) Parallel composition of two components. (c) Reliability block diagram for a TMR system obtained by a serial composition and parallel composition using single components multiple times. (d) Example of a non-serial/non-parallel system.

The serial composition of two components a and b, as it is shown in figure 1-12 (a), yields a component c that is operational if both components a and b are operational. The parallel composition of two components a and b, as it is shown in figure 1-12 (b), yields a component c that is operational as long as a or b are operational. The advantage of modeling a system in this way is the simple and fast computation of the reliability of the whole system based on the reliabilities known for the elementary components. For a serial composition as shown in (a) the reliability of component c is computed by

$$R_c(t) = R_a(t) \cdot R_b(t)\,.$$

And for a parallel system as shown in (b) the reliability of the component c is computed by:

$$R_c(t) = 1 - (1 - R_a(t)) \cdot (1 - R_b(t))\,.$$

Thereby, $(1 - R_a(t)) \cdot (1 - R_b(t))$ is the probability that both components a and b are faulty. Thus, $1 - (1 - R_a(t)) \cdot (1 - R_b(t))$ is the probability that both components are not simultaneously faulty, which means that at least one of them is functioning.

The drawback of such a reliability block diagram is its limited modeling power. For example, using the proposed composition rules, it is not possible to model a TMR system [116]. In order to overcome this limitation, various modifications were introduced. For example, components for modeling an M-of-N system may be

allowed. Then a TMR system including a voter can be modeled by a serial composition of a 2-of-3 system with reliability $R_{2,3}(t)$ and a voter component with reliability $R_{voter}(t)$:

$$R_{TMR} = R_{2,3}(t) \cdot R_{voter}(t) .$$

By allowing the incorporation of M-of-N systems into the model, additional semantic information is needed in the model. I.e., for N parallel components in the RBD the maximal allowed number of failing components must be known. In some extensions of the RBD model the multiple usage of components is allowed [116]. This allows for modeling a TMR system as shown in figure 1-12 (c), too.

RBDs may be also used to represent systems that are not built-up with the serial and parallel composition rule [94]. This yields non-serial/non-parallel system as shown in figure 1-12 (d). Please note that a path containing nodes a and d is not valid. For these systems the computation of the reliability becomes more complex, because the system is expanded about each module x that violates the serial/parallel composition rules. For such a module x a case-by-case analysis must be done. One new RBD is constructed for the case that x is faultless, and another RBD is constructed for the case that x is faulty [94]. Both cases are mutual exclusive. For this reason the obtained reliability functions for both RBDs can be added in order to obtain the reliability function for the original system. By performing this kind of expansion for each elementary component of the original RBD, the combinatorial approach described in section 1.1.4.1 is obtained.

Fault Trees

Fault trees are just another representation of reliability block diagrams, i.e., equivalence of both models can be proven [116]. The inner nodes of the fault tree either represent a conjunction or a disjunction of the Boolean values of their sons. Each leaf represents a single component of the system. Thereby, the Boolean value of a leaf corresponds to the operational state of the component represented by the leaf. Now the operational state of the system corresponds to the Boolean value of the root node, which is determined bottom-up, by evaluating the Boolean value of each inner node based on the Boolean values of its sons. It is obvious that the conjunction is equivalent to the serial composition in the reliability block diagram, and that the disjunction is equivalent to the parallel composition in the reliability block diagram. Thus, a fault tree just represents the composition of the RBD by making the application of serial and parallel composition rules explicitly visible. The repeated usage of components, as it is possible in reliability block diagrams, is modeled in fault trees by having multiple leafs representing the same component.

1.1.4.2 State-Space Models

In state space models, each fault state of the system is represented explicitly. Moreover, events – for example, a component becomes defect or becomes repaired – are modeled by state transitions. A widely used model is the Markov model, which is represented by a directed graph with nodes V and edges E. Each node $i \in V$ represents a particular fault state of the system, and it is labeled with $\lambda_i \in \mathbb{R}$. λ_i represents the constant rate with which system state i is left. The edges represent events that cause such a state transition. When the system leaves state i, then a transition into state $j \neq i$ with $(i,j) \in E$ occurs with probability p_{ij}. For this reason, each edge (i,j) is labeled with $p_{ij} \in \mathbb{R}$, $0 \leq p_{ij} \leq 1$. Because a state i may have various successor states, for each state i must hold:

$$\sum_{(i,j)\in E \text{ and } i\neq j} p_{ij} = 1\,.$$

Please note that p_{ij} only denotes the probability of entering state j, when state i was already left. Therefore, $\lambda_{ij} := \lambda_i \cdot p_{ij}$ is the rate of systems that will leave state i and enter state j. The following simple example is a Markov model of a TMR system without a voter. The TMR system is composed of three identical components, each of them having the constant failure rate λ.

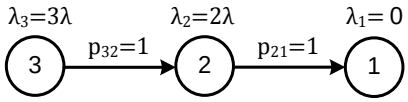


Figure 1-13: Markov model of a TMR system.

State 3 represents the system state in which all three components are operational. The reliability of such a system is given by $e^{-3\lambda t}$. Therefore, it has a constant failure rate of 3λ. If a failure occurs, then the system changes into state 2, representing a situation where 2 out of 3 components are operational. In this state the failure rate of the two remaining components is 2λ. In state 1 at most one component is operational, which is equivalent to a system failure. Because there is no repair function, the system will never leave state 1. In this model each state has at most one successor state. Therefore, each edge must be labeled with 1.

The semantic of the Markov model is given by a special stochastic process, which is an infinite number of random variables $X(t)$ that are indexed by continuous time t, $t \in \mathbb{R}$ and $t \geq 0$ [94]. The discrete state of the system at a particular time t is given by the value of $X(t) \in V$. Because V is a set of discrete states, there is a particular time for each state transition of the system. The infinite set $\{t_k \in \mathbb{R} |\ k \in \mathbb{N}\}$ should contain those points in time, when the state of the system changes. In order to derive from a Markov model the reliability function of the modeled system, the probability that the system is in state i at a particular time t must be known for each state $i \in V$. Let $P_i(t)$ denote this probability. The

important property of the Markov model, which simplifies its analysis, is that the probability of being in state $X(t_n)$ at time t_n only depends on the probability of being in a state $X(t_{n-1})$ at time t_{n-1}. This property is used in equation (1-21) for determining $P_i(t)$ by considering only the immediate predecessor states of state i:

$$\frac{dP_i(t)}{dt} = -\lambda_i P_i(t) + \sum_{j \neq i} \lambda_{ji} \cdot P_j(t)\,. \tag{1-21}$$

The deviation of $P_i(t)$ in equation (1-21) is the variation of the probability of being in state i. This variation is obtained by taking into account the rate of leaving state i, assuming that the system is in state i at time t (the first summand in equation (1-21)), and the rate of moving into state i from any of its predecessor states j, assuming the system is in state j at time t (second summand in equation (1-21)). Now, for each state i in the Markov model a differential equation is formed according to equation (1-21). This yields a set of differential equations, such that $P_i(t)$ can be determined for every state i. The following differential equations are obtained for the TMR model in figure 1-13:

$$\frac{dP_3(t)}{dt} = -3\lambda P_3(t)$$

$$\frac{dP_2(t)}{dt} = -2\lambda P_2(t) + 3\lambda P_3(t)$$

$$\frac{dP_1(t)}{dt} = 2\lambda P_2(t)$$

The solution of this set of differential equation yields for each function $P_i(t)$ the probability of being in state i at time t. Solving these differential equations with the initial conditions $P_3(t) = 1$ and $P_2(t) = P_1(t) = 0$ yields

$$\begin{aligned} P_3(t) &= e^{-3\lambda t} \\ P_2(t) &= 3 \cdot e^{-2\lambda t} - 3 \cdot e^{-3\lambda t} \\ P_1(t) &= -3 \cdot e^{-2\lambda t} + 2 \cdot e^{-3\lambda t}. \end{aligned}$$

Thereby the states 2 and 3 represent operational states of the TMR system. Thus, the probability that the system is in an operational state is the well known formula for TMR systems:

$$\begin{aligned} R_{TMR}(t) &= P_3(t) + P_2(t) \\ &= e^{-3\lambda t} + 3 \cdot e^{-2\lambda t} - 3 \cdot e^{-3\lambda t} \\ &= e^{-3\lambda t} + 3 \cdot e^{-2\lambda t} \cdot (1 - e^{-\lambda t}) \\ &= \left(e^{-\lambda t}\right)^3 + 3 \cdot \left(e^{-\lambda t}\right)^2 \cdot (1 - e^{-\lambda t}) \\ &= R^3(t) + 3 \cdot R^2(t) \cdot (1 - R(t)), \end{aligned}$$

whereby $R(t) = e^{-\lambda t}$ denotes the reliability of a single component in the TMR system with the constant failure rate λ.

This simple example has illustrated the usage of Markov models for modeling the reliability of systems, where all components are taken into operation at the same time $t_0 = 0$. This fact was model for the example in figure 1-13 by setting $\lambda_3 = 3\lambda$ in the model, which means that a fault may affect any of the three components in the initial state of the TMR system. However, in systems employing active hardware redundancy with cold standby the spare components are powered off in initial state. They are only taken into operation when another active component has failed. During the time, when the component is powered off, a negligible failure rate is assumed for this component. This situation can be modeled with Markov models, too, as shown in figure 1-14 (a).

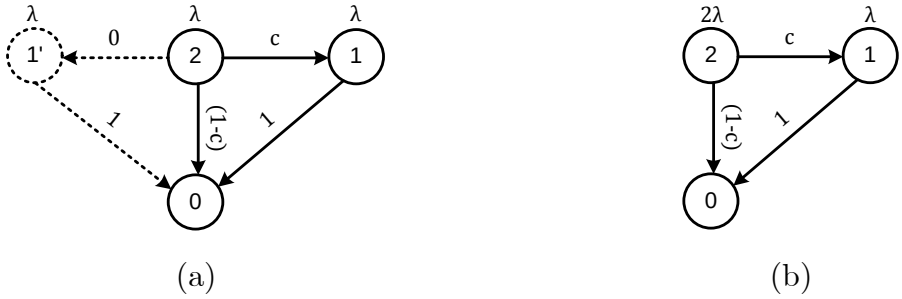


Figure 1-14: (a) Markov Model of a Fail-Stop System with cold standby adopted from [30, 94]. (b) Markov model for the Fail-Stop system from (a), but with hot standby.

The model in figure 1-14 (a) represents a fail-stop system composed of two processors. In order to detect a fault, the first processor performs some kind of self-checking; e.g. an acceptance test. If the acceptance test detects an error, then the first processor is taken out of operation, and it is assumed that the second processor can be taken into operation instantaneously. A switch that is used to select either one of the outputs of both processors is assumed to have a reliability of 1. The situation that not every fault of the first processor can be detected by acceptance tests is modeled by a fault coverage c with $0 \leq c \leq 1$, which is the probability that an occurred fault is detected, and the backup processor is successfully taken into operation. Only one of these two processors is taken into operation at time t_0. This initial situation is represented by state 2 in the Markov model. A failure within the first processor occurs with a constant failure rate λ. Thus, state 2 is left with probability λ. If the occurred fault is detected by the acceptance test, then the system moves into state 1 with probability c, i.e., the second processor is taken successfully into operation with probability c. With probability $\lambda \cdot (1\text{-}c)$ the system will leave state 2 and move to state 0, due to a fault that cannot be detected. The dotted state 1' represents the negligible situation that the unpowered processor has a fault. It is just shown for better

understanding the complete model and it can be removed from the model, because there is no chance to reach this state. For the Markov model in figure 1-14 (a) the following set of differential equations is derived:

$$\frac{dP_2(t)}{dt} = -\lambda P_2(t)$$

$$\frac{dP_1(t)}{dt} = \lambda c P_2(t) - \lambda P_1(t)$$

$$\frac{dP_0(t)}{dt} = \lambda(1-c) P_2(t) + \lambda P_1(t)$$

Solving this set of equations using the initial conditions $P_2(t) = 1$ and $P_1(t) = P_0(t) = 0$ yields

$$P_2(t) = e^{-\lambda t} \qquad P_1(t) = c\lambda t \cdot e^{-\lambda t} \qquad P_0(t) = 1 - P_1(t) - P_2(t) \tag{1-22}$$

Thus the reliability of the system is given as $R_{cold}(t) = P_2(t) + P_1(t)$. The same formula can be obtained by using the combinatorial approach as it was introduced in section 1.1.4.1. This is shown next, in order to clarify the difference between modeling hot- and cold-standby with combinatorial means.

In figure 1-15 $R(t) = e^{-\lambda t}$ of a single processor is shown, and the reliability of the fail-stop system with cold standby at time T should be determined.

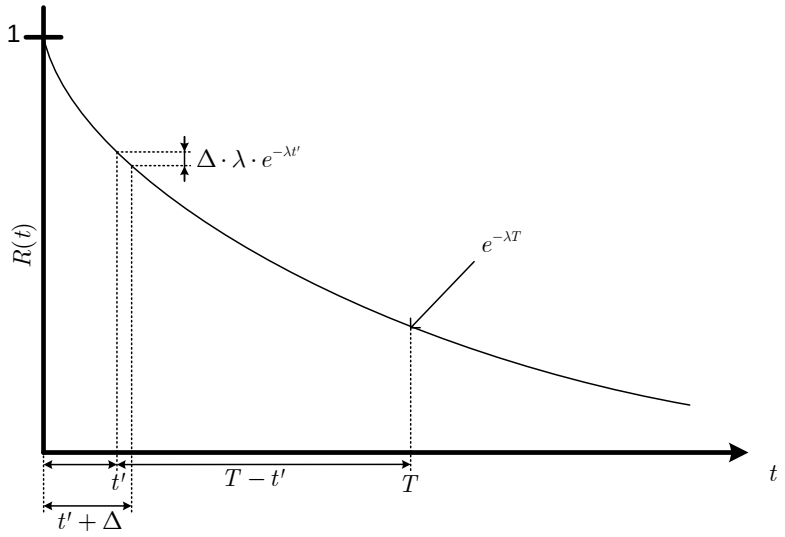


Figure 1-15: Reliability plot for a single processor of the fail-stop system modeled in figure 1-14.

It is obvious that the fail-stop system is operational at time T, if the first processor has survived until time T. This happens with probability $e^{-\lambda T}$ and corresponds to $P_2(T)$ in equation (1-22) (or state 2 in the Markov model). However, the fail-stop system is also operational, if the first processor p_1 has failed before T and the second processor p_2 was taken successfully into operation and is still operational at

time T. In order to obtain the probability for this scenario, the interval $[0, T]$ is divided into N very small time intervals Δ, i.e. $T = N \cdot \Delta$. Now consider an arbitrary time $t' = n \cdot \Delta$, with $0 \leq n < N$ and $n \in \mathbb{N}$ (see figure 1-15). The probability that p_1 has survived until time t' is $e^{-\lambda t'}$. In order to obtain the probability that a system will fail within the time interval $[t', t' + \Delta]$, the difference between $R(t')$ and $R(t' + \Delta)$ is computed. It is assumed that the function $e^{-\lambda t}$ is almost linear within the short interval $[t', t' + \Delta]$. The slope of the function R at time t' is $-\lambda \cdot e^{-\lambda t'}$. Thus, the probability that p_1 will fail within the interval $[t', t' + \Delta]$ is approximately $\Delta \cdot \lambda \cdot e^{-\lambda t'}$ (the sign in the deviation $-\lambda \cdot e^{-\lambda t'}$ is ignored, keeping in mind that the slope is negative). Furthermore, not every fault in p_1 may be detected by the acceptance test. This is modeled by the coverage factor c again. Thus, the probability that the second processor p_2 is successfully taken into operation in the time interval $[t', t' + \Delta]$ is $c \cdot \Delta \cdot \lambda \cdot e^{-\lambda t'}$. The probability that p_2, if taken into operation within the interval $[t', t' + \Delta]$, will survive until time T is $e^{-\lambda(T-t')}$. Putting both together yields the probability that p_2 is taken into operation within the interval $[t', t' + \Delta]$ and it will survive until time T as:

$$\begin{aligned} & c\Delta\lambda \cdot e^{-\lambda t'} \cdot e^{-\lambda(T-t')} \\ = \; & c\Delta\lambda \cdot e^{-\lambda t' - \lambda(T-t')} \\ = \; & c\Delta\lambda \cdot e^{-\lambda(t' + (T-t'))} \\ = \; & c\Delta\lambda \cdot e^{-\lambda T} \end{aligned} \tag{1-23}$$

Please note that equation (1-23) only takes into account a particular time t'. In order to obtain the probability that p_1 has been successfully replaced by p_2 at any time within the interval $[0, T]$, the consideration presented above for t' must be repeated for each $t' = n \cdot \Delta$, with $0 \leq n < N$. Now, recall that the interval $[0, T]$ was partitioned into N very small time intervals, each of them corresponding to a particular time t'. Because t' can be eliminated in equation (1-23), the probability that p_1 is successfully replaced by p_2 and p_2 will survive until time t is given by summing up the probabilities corresponding to each time interval:

$$\begin{aligned} & \sum_{t' \in \{t_i \mid 0 \leq i < N \text{ and } t_i = \Delta \cdot i\}} c\Delta\lambda \cdot e^{-\lambda T} \\ = \; & \sum_{i=0}^{N-1} c\Delta\lambda \cdot e^{-\lambda T} \\ = \; & N \cdot c\Delta\lambda \cdot e^{-\lambda T} \\ = \; & \frac{T}{\Delta} \cdot c\Delta\lambda \cdot e^{-\lambda T} = T \cdot c\lambda \cdot e^{-\lambda T} \end{aligned} \quad .$$

The obtained probability corresponds to the probability of being in state 1 in the Markov model.

The Markov model for the same system, but with hot standby, is shown in figure 1-14 (b). There, state 2 is the initial state, which is left with probability 2λ, because both processors are taken into operation at time t_0. Again, c is the probability that the occurred fault is detected by the acceptance test and (1-c) is the probabilities that the occurred fault is not detected by the acceptance. The reliability $R_{hot}(T)$ of such a system is easily obtained by combinatorial arguments, similar to a TMR system: The system is operational at time T, if

- both processors are faultless or
- the first processor is faultless and the second one is faulty or
- the second processor is faultless and the first one is faulty.

The probability that both processors are working at time T is $e^{-\lambda \cdot T} \cdot e^{-\lambda \cdot T}$, assuming a constant failure rate λ. The probability that one processor is faulty and the other processor is faultless is given by $2 \cdot e^{-\lambda \cdot T} \cdot (1 - e^{-\lambda \cdot T})$. Taking into account that the fault of a single processor is detected with probability c, the system reliability is

$$R_{hot}(T) = e^{-\lambda T} \cdot e^{-\lambda T} + 2 \cdot c \cdot e^{-\lambda T} \cdot (1 - e^{-\lambda T}).$$

A plot of the reliability of the hot and cold standby system is shown in figure 1-16. It can be observed that the reliability of the hot standby system is lower than the reliability of the cold standby system.

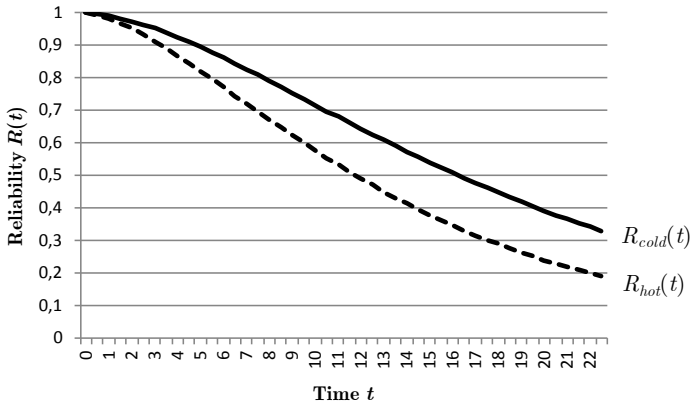


Figure 1-16: Plot of the reliability functions of the hot and cold standby systems from figure 1-14 (a) and (b).

This example has illustrated how to model permanent faults in reconfigurable systems with cold and hot standby. Modeling of such systems is possible with Markov models, but also with combinatorial approaches. Combinatorial approaches will become inconvenient for more complex replacement schemes as they are used for example in cold standby systems. Reconfigurable systems with hot standby, where all components of the system will age simultaneously, can be modeled using either Markov models or combinatorial approaches. However, the combinatorial approach is easier to use in these cases. The self-repair approach

proposed in this thesis uses hot standby. For this reason the reliability is computed with combinatorial approaches.

1.2 Hardware Redundancy in Processors

Hardware redundancy, as it is needed for implementing fault tolerance, is inherently available in many processors. However, the redundancy in processors is typically used for improving the performance by performing computations in parallel. According to the classification of Flynn there can be parallel data streams and parallel instruction streams [61]. Even for embedded systems it is common today to have multiple instruction and multiple data streams by having multi-core architectures. This provides coarse grained parallelism at thread and task level. But even single instruction and single data stream processors provide finer grained parallelism at instruction level. Thereby, the parallel execution of instructions takes place either by a spatial parallel execution (superscalarity) or by a temporal parallel execution (pipelining). For a spatial parallel execution, redundant hardware must be provided, as it is also needed for fault tolerance. For pipelining the available hardware is better utilized than in a non-pipelined architecture. I.e., in the ideal case, the execution of operations is divided into independent phases that use mutual exclusive parts of the processors hardware. When multiple operations are executed parallel, then they use these parts in a time multiplexed manner, such that each operation uses another part of the processor at the same time. According to these two types of parallelism, single instruction stream processors can be further classified into *scalar/superscalar* and *pipelined/non-pipelined* microprocessors. The execution scheme of instructions in these four types of processors is shown in figure 1-17, assuming that the execution of an instruction is divided into the four phases fetch (FE), decode (DE), execute (EX), and write-back (WB).

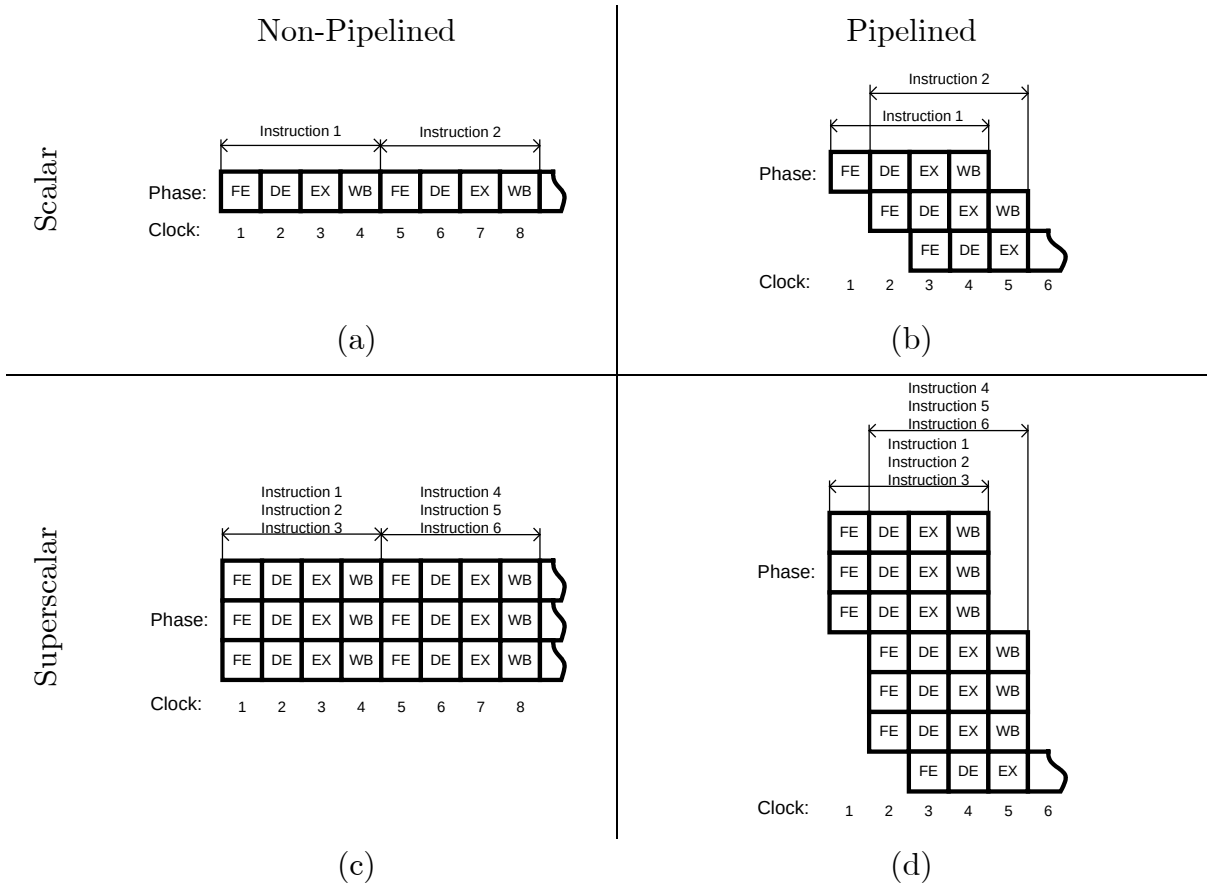


Figure 1-17: Classification of processors according to their type of provided parallelism.

In a non-pipelined scalar processor architecture the execution of an instruction sequence is done instruction by instruction (see figure 1-17 (a)). In a pipelined scalar processor architecture the execution phases overlap (see figure 1-17 (b)). In both processor architectures most resources are not redundantly available. In order to obtain fault tolerant scalar processors, redundancy must be added explicitly. Non-pipelined superscalar processors (see figure 1-17 (c)) and pipelined superscalar processors (see figure 1-17 (d)) can have multiple operations in the same execution phase. For this reason the resources for implementing these execution phases must be organized redundantly in different *computation domains*. This inherently available redundancy can be also used for making non-fault tolerant superscalar processors fault tolerant. Two scheduling policies exist for superscalar processors, when assigning operations from a single instruction stream to a particular computation domain:

- dynamic scheduling and
- static scheduling.

When dynamic scheduling is used, then the processor dynamically assigns each operation to a free computation domain during the execution of the application. This requires a dependency analysis that must be performed by the processor,

which increases the complexity and size of the control logic of the core. The software has no control about the computation domain that is used for executing a particular operation. If the usage of a defect computation domain should be avoided, as it is done in active hardware redundancy, then the administration of the redundant domains must be done in hardware by the processor itself, i.e., hardware administrated hardware redundancy must be used.

When static scheduling is used, then the processor is relieved from dependency analysis. This analysis is done statically by the compiler that must schedule the operations in such a way that the processor knows exactly which operations can be executed in parallel. An example of such a processor is the Itanium processor from Intel [155, 156][7]. This processor must not check for data dependencies, but it determines dynamically a computation domain; i.e., the binding of operations to computation domains is computed dynamically. *Very long instruction word* processors (*VLIW processors*) neither perform dynamic scheduling nor dynamic binding. This class of processor architectures has been proposed first by Fisher in the beginning of the 1980's together with appropriate scheduling techniques for parallelizing operations across basic block boundaries [59, 60]. The charm of this architecture is that complex administrative tasks like dependency checking are shifted to the compiler, which simplifies the control logic of this processor architecture very much. Operations that must be executed in parallel are grouped together by the compiler into a single very long instruction word. The binding of operations is either explicitly or implicitly coded in the instruction word. In contrast to processors that provide dynamic scheduling and binding, this allows for a complete software-based administration of redundant computation domains, i.e., software-administrated hardware-redundancy may be used. This property constitutes the base for the application of the software-based self-repair techniques proposed in chapter 3 and follows the VLIW paradigm of performing complex administrative tasks in software.

1.3 Summary

In this chapter the basic notations for dependable systems were introduced by considering a classification of means and impairments that affect the dependability of a system either in a positive or negative manner. An overview about fundamental fault tolerance methods that are based on redundancy for handling either transient or permanent faults in the field was given. Beside the well known classification of redundancy into hardware redundancy, software redundancy, time redundancy, and data redundancy, an orthogonal classification of redundancy was

[7] Also referred to as Explicitly Parallel Instruction Computing (EPIC) [156].

introduced. This orthogonal classification distinguishes between the system layers where the administration of the redundant resources takes place, and allows for a classification into software-administrated and hardware-administrated redundancy.

The self-repair methods presented in this thesis will fall into the category of active hardware redundancy methods, whereby in most cases the hardware redundancy will be administrated in software. The term self-repair was defined as the reconfiguration step needed for active redundancy, no matter whether this reconfiguration step is related with graceful performance degradation or not. The self-repair methods will primarily target at permanent faults that will occur during the operational phase of the system, no matter which physical mechanism has caused these permanent faults. For the evaluation of the proposed methods the reliability was introduced. Stochastical methods for computing the reliability of a fault tolerant system, whose components may fail due to permanent faults, were introduced.

Although in active hardware redundancy approaches error detection and fault localization precedes the self-repair phase, in this thesis, first, various self-repair methods will be considered. By developing and analyzing these methods, it becomes clear at which granularity level they can work. This allows for developing a diagnostic self-test method that fits very well to the presented self-repair methods. This guarantees the required diagnostic resolution needed for the self-repair process. I.e., it is avoided that a lot of time and resources are spend for the diagnosis of faults that cannot be handled by the subsequent self-repair process.

Chapter 2

Hardware-Based Self-Repair

A single statically scheduled superscalar processor architecture is used throughout this thesis in order to illustrate the proposed self-test and self-repair concepts. The processor architecture is named VARP, which means VLIW Architecture for Research Purposes. Different self-repair schemes presented in chapter 2 to chapter 4 for that processor model require different extensions of the processor model. First, the basic non-fault-tolerant VARP processor is presented in section 2.2. Then, the non-fault tolerant VARP processor is made fault tolerant by applying hardware-administrated active hardware redundancy in section 2.3. For this reason the redundantly available computation domains are used for replacing each other. Because the redundant hardware components are administrated by hardware, it is a hardware-based self-repair approach. The administration component is a simple hardware extension that changes on-line the binding of operations to computational domains. This approach is able to handle multiple permanent faults in the computational domains of the VARP processor, but at the cost of performance degradation for most fault situations. Reliability and performance estimations are presented in section 2.4. Both, the achieved reliability improvement and the performance degradation are used as reference values for the comparison with the software-based methods presented later in chapter 3.

2.1 Related Work

In order to built fault tolerant processors, codes and passive hardware redundancy techniques like DMR and TMR for error detection and fault masking are state of the art. They are used in various academic [11, 145] and commercial processor based systems [66, 172]. According to the classification in figure 1-9, the redundancy in those systems is most likely managed at hardware-level, in particular at architecture- and register transfer-level. Gaisler describes in [66] how the redundancy is employed and managed at register transfer level for making the memory elements of the LEON-FT core fault tolerant. For example, the cache memory is protected by parity bits. A detected error is corrected by accessing the original value in the main memory. TMR is used for fault masking in pipeline- and processor-registers. IBMs S/390 G5 processor uses codes and hardware-administrated DMR techniques for recovering from temporary faults [172], too. DMR is also used at architecture level by having full duplicates of the instruction decode- and execute-units and performing cross-checks of the results. Moreover, check pointing of these units is supported by a special recovery unit for recovering from temporary faults. Also recovery from permanent faults is supported by active hardware redundancy. For this purpose, spare cores are available in the system, and transfer of checkpoints from a faulty core to a spare core is supported by hardware. Similar techniques were already proposed a few years earlier by Franklin for superscalar dynamically scheduled processors. In [62] Franklin discusses various strategies for detecting temporary and permanent faults in the execution units. The basic idea is to execute each operation twice by using the inherent hardware redundancy in superscalar processors. Various strategies for duplicating the operations are discussed. For example, the operation may be duplicated either by the hardware scheduler of the processor or by each execution unit itself. In the former case also permanent faults in the execution units can be detected. In [63] Franklin presents more ideas for detecting and handling faults in several other components of a superscalar processor, e.g. data- and address-errors in the instruction cache, errors in the fetch- and decode units, errors in the dynamic scheduler, and errors in the register file. Thereby the proposed fault tolerance methods are based on codes or re-execution of operations; i.e., information and time redundancy is employed and managed at hardware-level. Unfortunately, no results of an implementation in a real processor are available.

Passive hardware-redundancy is also employed in processors at the finer-grained gate level. Mitra et al. have proposed fault masking techniques at gate level that can handle single event upsets in scan flip-flops that occur when the clock is low [122]. Scan flip-flops are built from two master-slave flip-flops, such that during normal operation information can be stored redundantly. Fault masking is

obtained by using a Muller-C element at the outputs of both flip-flops. A Muller-C element has two data inputs and a single data output. If both inputs have the same value, then the output will have this value, too. But when both inputs differ, then the Muller-C element keeps the latest output value for which both inputs were equal. When the stored information in the slave-latch of one of the two flip-flops is inverted by a SEU, then the Muller-C element will keep the correct value. The error is corrected when the clock rises to high by overwriting the faulty value with the value from the corresponding master latch. This reduces the probability for the occurrence of SEU, but cannot totally eliminate them, because the flip-flop cannot handle SEU that affect a master latch when the clock is high. In this situation the error is detected, but recovery must be done at a higher level of administration, which requires a global error signal for each scan flip-flop. Redundant latches for error detection at gate level are used in the RAZOR flip-flop [56], too. However, the RAZOR flip-flop has been designed for the detection of delay faults in processor pipelines. For this reason the redundant shadow latch will buffer a slightly delayed value from the combinatorial input of the previous pipeline stage. If the correct value arrives too late, then both values in the original flip-flop and in the shadow latch differ, which can be detected. In this case, an extra clock cycle is needed for error correction, i.e., moving the correct value from the shadow latch into the original flip-flop. However, this local delay must be propagated to all other flip-flops in the processor, which also requires a global administration scheme. Moreover, the protected memory elements in these approaches have the same data input. SET affecting the combinatorial logic driving this input will not be detected by these methods. For this reason a duplication of the combinatorial circuitry is considered in [121]. Reduction of the costs for duplicating the combinatorial circuitry is achieved in [173] by using a predictor circuit instead of a duplicated one. The predictor circuit generates a signature that is compared with the signature generated from the result of the combinatorial logic. In case of a mismatch, due to a SET, a global error signal is generated that forces all slave latches to keep their latest correct value, until the SET disappears. A major draw-back of fine-grained local fault handling is the need for a complex global administration scheme. Moreover it is obvious that such passive hardware-redundancy techniques can handle temporary faults only, because the error recovery mechanism relies on the assumption that the fault, which was the cause of the error, disappears after a short period of time.

Because permanent faults do not disappear, they are better handled by active hardware redundancy that takes the faulty component out of operation. An active hardware-redundancy scheme for handling permanent faults at switch- and gate-level was proposed from Kothe et al. in [96]. I.e., all gates are replicated, and additional transistors are used for isolating gates from supply voltage, in order to

handle shorts. However, such a fine-grained replication scheme requires too many administrative overhead for local switches. For this reason, the same group has proposed a reconfiguration scheme, where reconfigurable logic blocks are used as redundant components. The size of these reconfigurable logic blocks may range from simple gates to more complex combinatorial logic like adders and ALUs [90]. By this, the hardware-overhead for local switches is significantly reduced. Moreover, if the reconfiguration is done in off-line mode, then a complex global administration scheme for controlling the timing is not needed.

A similar and very natural way for implementing active hardware redundancy that is administrated by hardware is the usage of field programmable gate arrays (FPGAs), which are composed of configurable logic blocks, and provide hardware-redundancy at that granularity level. Processors may be mapped as soft-cores into a FPGA [58, 85, 125]. By mapping a soft core twice in a FPGA, DMR can be used for handling transient faults at system level. Such a technique has been used in [58] for the LEON core. Moreover, when a permanent fault in one of the configurable logic blocks of the FPGA affects the soft core, then a another configuration of the FPGA can be used, such that the soft core is no longer mapped into the faulty area of the FPGA [125]. In [77] it was shown that a computation of a new fine-grained configuration based on configurable logic blocks becomes very time consuming in the field due to the modified routing [77]. For this reason precompiled configuration schemes are used for example in [120] and [174] at the cost of larger backup areas. Moreover, a soft-core may be used for administration of hardware redundancy as well as configurations with multiple FPGAs. By this FPGAs can be used for reconfiguring each other [120]. Thus, FPGAs provide high flexibility but at the expense of hardware overhead, lower performance, and extra power. For example, implementing the VARP processor in a Xilinx Virtex 6 and Virtex 7 FPGA allows for a maximal clock rate of approximately 200 MHz [136]. The implementation of the VARP processor as a hard-core with a comparable feature size allows for more than 500 MHz as it will be shown by the synthesis results in section 2.2.3. In order to build high performance, low-power and low-area embedded processor based systems it is more beneficial to implement the processor as a hard-core with some specific reconfiguration facilities.

This has been done for the dynamically scheduled superscalar Alpha 21264 microprocessor [67] for yield improvement. Three fundamental hardware-based administration schemes for active hardware fault tolerance were proposed for this processor from Shivakumar et al. in [168]. The administration schemes were tailored to dedicated component types like queues, register arrays, and redundant execution units that can be typically found in this type of processor architecture. The basic idea is to use the control logic of these components for fault handling in

them. By this, the amount of extra control logic for administrating the redundancy should be reduced. The first administration scheme is for component level replication and can be used when components are available redundantly for high performance, but the system remains functional as long as at least one of them is operational. A permanent fault in one of these components is then treaded as if this component is busy all the time. This administration scheme applies in particular to the execution units in a dynamically scheduled processor and is comparable to the one proposed by Franklin in [63]. The second administration scheme for array redundancy is applied to components that are organized as bit fields, and faults in some of the bit-fields can be handled by re-routing the accesses to the faulty components to non-faulty components. This, for example, applies to register files, where the usage of faulty registers is avoided by hardware-based register renaming using some spare registers. Finally, the third administration scheme for dynamic queue redundancy is applied to queues in the processor as they are used, for example, in operation queues and reorder buffers. It is proposed to handle faults in some of the queue elements with the help of the control logic of the queue by treating them as occupied. Similar mechanisms were used by Bower et al. in [29] for handling hard faults in various components of the same processor architecture, when these faults occur during the operational phase in the field. Fault detection was implemented by using DIVA checkers [12]. DIVA checkers are small units added into the commit-stage of the pipeline of a superscalar processor[8]. They are used for computing the result that should be committed a second time. It is claimed that DIVA checkers are implemented in a more robust way than the rest of the core, such that a detected fault is accounted to the original units of the superscalar processor.

Most of the proposed administration schemes for redundancy in superscalar dynamically scheduled processors target for data path elements only. However, in the Alpha-processor the size of the data path (integer plus floating-point slots) is approximately five times the size of the control logic. In a comparable VARP processor the ratio of data path (size of all four slots) to the control logic is estimated to be around 126. It is obvious that this ratio in a statically scheduled processor is much higher, because the control logic in statically scheduled processors can be much simpler than in dynamically scheduled processors, because the hardware-based optimizations in dynamically scheduled processors come at the expense of more control logic. However, fault tolerance is based on redundancy, and redundancy is hardly found in irregular control logic. For this reason the irregular control logic cannot be protected by active hardware redundancy, except

[8] Originally DIVA checker were used for dynamic verification of the components of a superscalar processor. In this way design fault should be detected and corrected on-line [12].

redundancy is explicitly added to the control logic. Hence, fault tolerance techniques that try to employ the inherently available redundancy in processors are more beneficial for statically scheduled processors than for dynamically scheduled processors, because in statically scheduled processors a larger portion of the core area is occupied from the data path that provides inherently available redundancy. For this reason, techniques for hardware-based administration of the redundancy in statically scheduled processors are considered now.

An active hardware redundancy scheme for pipelined scalar cores was presented in [149]. Because a single scalar core does not provide hardware redundancy, several of them are composed to a multi-core system. Faults in two or more processors are handled by cannibalizing a faultless core. I.e., the faulty cores replace their own faulty pipeline stage with the functioning pipeline stage of the cannibalized core. Chen et al. has proposed in [43] an active hardware redundancy scheme for a statically scheduled VLIW processor. This superscalar processor inherently contains redundant hardware in its data path. The architecture of the VLIW processor is very similar to one of the VARP processor. The usage of a permanently faulty component in the VLIW is avoided by routing operations that where fetched into a particular execution domain of the VLIW into another execution domain. For this reason, a hardware scheduler is integrated into the execution stage of the data path that can change dynamically the binding of operations. Furthermore, the results of the execution units can be compared in hardware. Faults are detected by a concurrent execution of operations and comparison of their results. Thereby the scheduler allows for a duplication and triplication of operations. If necessary, an operation is re-executed in order to localize the faulty unit. This approach is able to detect temporary and permanent faults in execution units. Permanent faults can be localized on-line and the usage of the faulty execution units is avoided. A similar approach is presented by Shyam in [170] for a VLIW architecture. There, the reconfiguration logic is integrated in the execution stage, too, in order to avoid the usage of a faulty execution unit of the processor. A fault is detected on-line by a special checker that checks the correct execution of operations by computing only a signature for the result, using a small watchdog-ALU. However, in [43] and [170] the scheduler respectively the reconfiguration logic are integrated into the execution stage. This means that faults that appear in a significantly large portion of the data path cannot be handled. For instance, faults in the decode stage of the processors will not be detected. At least the integration of the reconfiguration logic into an earlier pipeline stage is possible, but this requires a more complex administration scheme, especially for the recovery process after error detection, which finally increases significantly the size of the control logic.

A hardware-based reconfiguration scheme for VLIW processors that covers also faults in the decode-stage of the pipeline is proposed by Koal et al. in [92]. Spare components were added to the data path and encapsulated by a multiplexer network for reconfiguration purposes. Error detection and fault handling is done off-line. By this, a recovery scheme after error detection is not needed, because faults are detected off-line by preemptive error detection. I.e., after startup, a software-based self-test is performed. When this test fails, another configuration is tried and the self-test is repeated. In this way all feasible configurations are systematically applied until a functioning one is found. Then the user application is launched. Such a reconfiguration scheme can be only applied at a coarse-grained administration level; e.g. slot or execution unit level. Otherwise, the number of feasible configurations becomes too large.

The concept of preemptive error detection in off-line mode is also assumed for the VARP processor in the sub-sequent sections. Moreover, the integration of a hardware-based re-scheduler in an early pipeline stage is demonstrated that dynamically allocates operations to other execution domains.

2.2 The non-Fault Tolerant VARP Processor

The VARP processor is a simple and self-developed VLIW processor. According to the classification of Flynn [61], it is a single instruction stream and single data stream (SISD) architecture. I.e., only a single control flow is handled by the processor and only a single data memory-access operation can be executed each time. Parallelism is provided at instruction level. The VARP processor has a four stage pipeline and provides redundant resources due to its superscalar organized data path. Operations must be scheduled statically by the compiler. The processor is a Harvard-architecture, i.e., it has separated busses and memories for data and program code. The advantage is two-fold: First, data and program memory can have different bit widths. This allows for an efficient instruction coding, because the instruction word length does not depend on the data word length. Second, some resource hazards in the pipeline can be avoided. In the Princeton-architecture[9], which has a shared memory for data and program code, such hazards will occur, because fetching a new instruction from the program memory may interfere with a memory-access operation. On the other hand, it is very simple to employ self-modifying program code in a Princeton-architecture, due to the fact that each program has access to the data memory, and the data memory also contains the program code. This principle of self-modifying code is used for the software-based self-repair method proposed in chapter 3. Although a Harvard-

[9] Also known as von-Neumann-architecture.

architectures may not support this feature in such a natural way as the Princeton-architecture does, the VARP-processors uses the Harvard-architecture, because this allows for avoiding hazards caused by the instruction fetch and memory-access operation. A schematic picture of the VARP processor is shown in figure 2-1. The implementation used throughout this thesis has a 16-bit data path. However, other versions, for example with a 32-bit data path, are also available, because various parameters of this processor architecture, including data path size, can be scaled easily.

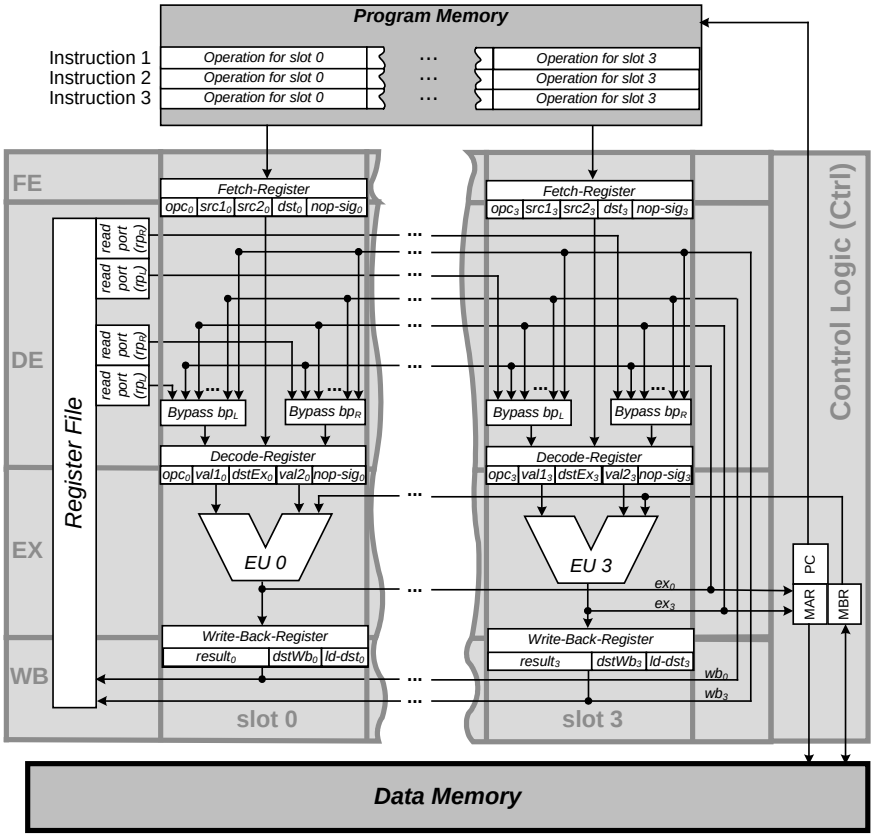


Figure 2-1: The VARP Processor.

Vertically the data path is divided into four computation domains, which are called *slot 0* to *slot 3*. Each slot is composed of a fetch register, two *read ports* (rp_L and rp_R), two *bypasses* (bp_L and bp_R), a decode register, an execution unit, and a write back register. The register file is shared by all slots. It contains 64 general purpose registers named *r0* to *r63*. The program counter (PC) holds the address of the next instruction to be fetched from the program memory. Each instruction in the program memory has a bit width of 104 bits. It contains 4 operations, whereby each operation is encoded into 26 bits. The encoding of an instruction is shown in figure 2-2.

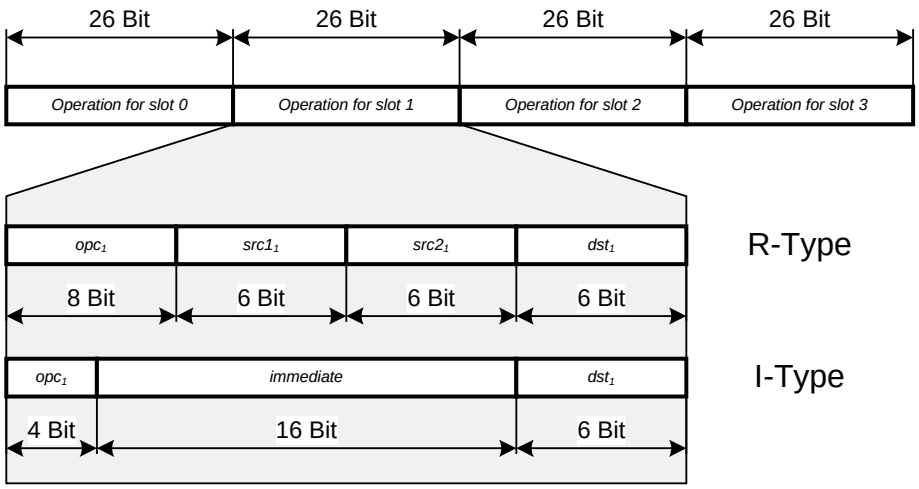


Figure 2-2: Encoding of a VARP instruction.

An instruction is composed of four operations. Figure 2-2 also shows their grouping into a single instruction. Which operation is executed by which slot is given by the relative position of an operation within an instruction. The coding of operations is very similar to the coding in a MIPS-like architecture [80]. Two types of operation encoding are used: *R-Type* and *I-Type*. Operations of the R-type encode two source register numbers (*src1* and *src2*) and one destination register number (*dst*). Eight bits are left for encoding the *opcode* (*opc*) of an operation. Operations of I-Type are used for encoding operations with a 16-bit immediate value. The opcode bit-field of I-Type operations has only 4 bits. For both operation types the upper four bits of the opcode encode the operation group. Their meaning is shown in table 2-1. A full description of the instruction set architecture is given in Appendix A.

Group	Meaning
0	Encodes up to 16 R-Type operations: Among them are a *NOP* (*No OPeration*) and *load*/*store*-operations for memory access. The VARP processor is a load/store-architecture, where data memory access is only possible via load/store-operations.
1	Encodes 16 conditional and unconditional jump operations of R-Type. Testing of a *carry*-, *negative*-, *zero*-, and *overflow*-flag is supported. Any general purpose register can serve as flag-register. Flags are explicitly set by the *cmp*-operation (see Appendix A). The branch target address is taken from a *src1*-register.
12	16 arithmetic and logical operations are encoded using the R-Type format.
2 to 11 14 and 15	Each group encodes exactly one operation of the I-Type. Among them are the load-constant operation (*ldc*-operation) and conditional/unconditional jumps where the target address is encoded as immediate value.
13	Unused.

Table 2-1: Opcode-grouping in the VARP instruction set architecture.

Horizontally the data path is divided into the four pipeline stages fetch, decode, execute, and write-back. They are organized very similar to the ones of a MIPS-like processor [80]:

- Fetch (FE): The instruction is loaded from the program memory.
- Decode (DE): The instruction is decoded into control signals, if necessary operands are read from the register file.
- Execute (EX): The required operation is executed by an *execution unit* (*EU*). For this purpose each execution unit contains various *operators*, for example adder, multiplier, etc.
- Write-Back (WB): The result is written back to the specified processor register.

When an instruction is fetched from the program memory, then each operation is stored in the fetch register of their corresponding slot k. Thereby the values of the bit fields of each operation (see figure 2-2) are stored into the corresponding bit fields of the fetch register (see figure 2-1). The bit field $nop\text{-}Sig_k$ in figure 2-1 is set to 1, if and only if the fetched operation for slot k is a *NOP*. This signal is used in the execution stage for generating the load-signals for the destination registers. All fetched operations of one instruction are processed in lock-step manner simultaneously in the processor pipeline.

During the decode stage the register values are provided. For this reason the values of the bit fields $src1_k$ and $src2_k$ in the fetch register of slot k are used as control signals for the left and right read port of slot k. The functions of the left read port (rp_L) and right read port (rp_R) of slot k are formally given by

$$rp_L(src1_k, r_0, r_1, ..., r_{63}) := r_{src1_k} \text{ and } rp_R(src2_k, r_0, r_1, ..., r_{63}) := r_{src2_k},$$

where *r0* to *r63* are the values of the registers in the register file. The left bypass (bp_L) and right bypass (bp_R) of a slot are used to select either the output value of the corresponding read port, or, if necessary, a value from the execution or write-back stage of the processor. The values generated in the execution stage are named ex_0, ..., ex_3, and the values from the write-back stage are named wb_0, ..., wb_3 (see figure 2-1). Then the function of the left bypass is given by

$$bp_L(src1, r_{src1}, ex_0, ..., ex_3, wb_0, ... wb_3, dstEx_0, ... dstEx_3, dstWb_0, ... dstWb_3, vEx_0, ..., vEx_3, vWb_0, ... vWb_3) := \begin{cases} wb_i, \text{ if } \exists i : dstWb_i = src1 \wedge vWb_i \wedge \\ \qquad \forall j \in \{0,1,2,3\} : dstEx_j \neq src1 \\ ex_i, \text{ if } \exists i : dstEx_i = src1 \wedge vEx_i \\ r_{src1}, \text{ otherwise} \end{cases}.$$

Thereby, r_{src1} is the register value provided by the corresponding read port rp_L. $dstEx_0$,...,$dstEx_3$ and $dstWb_0$,...,$dstWb_3$ are the destination register addresses of the

corresponding values ex_0,...,ex_3 and wb_0,...,wb_3. vEx_0,...,vEx_3. vWb_0,...,vWb_3 are the valid bits of these values. The value that is selected by the left bypass in slot k is moved into the bit field $val1_k$ of the corresponding decode register and the value selected by the right bypass is moved into the bit field $val2_k$. Furthermore, the values of the bit fields opc_k and dst_k of the fetch register are moved during the decode stage into the bit fields opc_k and $dstEx_k$ of the decode register.

The operation encoded in the bit field opc_k of the decode register is executed by the execution unit (EU) during the execution stage. Thereby $val1_k$ is used as left and $val2_k$ as right operand of the operation. The result of an arithmetic or logic operation in EU k is stored in the write-back register of slot k (bit field $result_k$) together with the number of the destination register (bit field $dstWb_k$). Beside arithmetic and logic operations, the EUs also perform branch and memory operations. For this purpose the EU generates appropriate values that can be loaded into the registers *PC*, *MAR* and *MBR* of the control path. All execution units are homogeneous. I.e., each execution unit is capable of executing branch and memory operations. This provides the required redundancy for self-repair purposes. Please note that it is the task of the compiler to make sure that at most one memory-operation and at most one branch operation is executed at the same time[10]. Moreover, execution unit k generates the load-signals for slot k ($ld\text{-}sig_k$) for all feasible target registers. These four load-signals are:

- $ld\text{-}dst_k$: this bit is one, if the operation generates a result that should be written into the register encoded in the bit field $dstWb_k$. The *ld-dst* signals of all slots are also used as valid-bits *vEx* and *vWB* for the bypass.
- $ld\text{-}pc_k$: this bit is one, if the executed the operation is a branch operation that has generated a new value for the PC, meaning that the program counter must be loaded with the result generated by execution unit k.
- $ld\text{-}mar_k$: this bit is one, if the result of the execution unit must be loaded into the memory address register (*MAR-register*). I.e., the executed operation has generated a memory address.
- $ld\text{-}mbr_k$: this bit is one, if the result of the execution unit must be loaded into the memory buffer register (*MBR-register*).

Please note that the *nop*-signal from the decode register is used for masking all of these load-signals. By this, a fault in the fetch- or decode-register that changes the opcode of a fetched NOP into the opcode of another operation, will not cause writing into any register.

[10] In principle the processor architecture supports multiple branch-operations in a single instruction. But, then the branch conditions must be mutual exclusive.

The instruction set of the processor is defined in such a way that each operation is executed within one clock cycle by an execution unit. For arithmetic operations, bit operations, and branch operations this can be done without any difficulty. However, a data memory access is assumed to take two clock cycles. Data memory access is either done by a load- (reading from memory) or by a store-operation (writing to the memory). Both operations are carried out in the execution stage. A dedicated memory stage, as for example in the MIPS-architecture, is not required, because the VARP architecture does not support address arithmetic. In order to perform a two clock cycle memory access with single cycle operations, the load- and store-operations are divided into two separated operations shown in figure 2-3.

```
ld0 [Rx],Ry        st0 Ry,[Rx]
ld1 [Rx],Ry        st1 Ry,[Rx]
```

(a) (b)

Figure 2-3: (a) Memory load-operation. (b) Memory store-operation.

The *ld0*-operation moves during the EX-stage the value of register x into the memory address register (MAR). The expected value is provided by the data memory one clock cycle later as input to all execution units. The *ld1*-operation, which is executed after the *ld0*-operation, stores this value into the write-back register. In a similar way the *st0*-operation moves the address and data value into the MAR and MBR. The *st1*-operation finally resets some control signals of the data memory. Please note that the *ld0*-, *ld1*-, *st0*-, and *st1*-operations may be executed in different slots.

Due to the organization of the pipeline and the used Harvard-architecture, neither resource nor data hazards will occur in the pipeline. Control hazards, due to a taken branch, are solved by discarding the instructions in the fetch and decode stage of the pipeline. When a branch is not taken, then the correct instructions were already fetched, and the operation of the pipeline simply continues.

2.2.1 Processor Model

In [157] a formal model of a VARP-like processor has been introduced for design space exploration. A simplified version of that model is now used for introducing a fault state model for the VARP processor[11]. An instance of the 16-bit VARP processor is characterized by the parameters (R, N, $\mathcal{O}$), where:

[11] The model is simplified in that sense that the VARP processor is a non-clustered homogeneous VLIW architecture; i.e., only a single centralized register file is provided and all execution units support the same operations, provided that there is no fault in an execution unit.

- $R \in \mathbb{N}$ is the number of registers in the register file.
- $N \in \mathbb{N}$ is the number of slots.
- $\mathcal{O}$ is the set of operations supported by each execution unit.

Now the components of a VARP processor, as it is shown in figure 2-1, are represented by the following sets. The set $REGS \subseteq \mathbb{N}$ with

$$REGS := \{0,\ldots,R\text{–}1\}$$

is the set of all registers in the register file. The set $SLOTS \subseteq \mathbb{N}$ with

$$SLOTS := \{0,\ldots,N\text{–}1\}$$

is the set of all slots. Each slot k is composed of two read ports, two bypasses, three pipeline registers and a single execution unit. The execution units are represented by the set $EUS \subseteq \mathbb{N}$. It is equal to the set of slots:

$$EUS := SLOTS.$$

Execution unit k belongs to slot k. A VARP processor with N slots has $2{\cdot}N$ read ports and $2{\cdot}N$ bypasses. Therefore read ports are represented by the set $RPS \subseteq \mathbb{N}$ and bypasses are represented by the set $BPS \subseteq \mathbb{N}$, whereby

$$RPS = BPS := \{0,\ldots,2{\cdot}N\text{–}1\}.$$

Thereby, read port/bypass $2{\cdot}k$ is called left read port/bypass of slot k and provides the left operand for execution unit k. Read port/bypass $2{\cdot}k{+}1$ is called right read port/bypass of slot k and provides the right operand for execution unit k.

The fault states of these components are specified by using various *fault state functions*. The fault state function

$$fsPipe : SLOTS \rightarrow \{0,1\}$$

reflects the fault state of the pipeline registers of a slot. Thereby $fsPipe(k) = 1$ means that the pipeline registers of slot k are faultless. If $fsPipe(k) = 0$, then there is a fault in a pipeline register such that the operation encoding is changed by this fault. The fault-state of an execution unit is expressed by the function

$$fsEU : \text{EUS} \times \mathcal{O} \text{ - } \{\text{NOP}\} \rightarrow \{0,1\},$$

where $fsEU(k,t) = 1$, if and only if an operation of type $t \in \mathcal{O}$ - {NOP} is executed correctly by execution unit k. For example, suppose the execution unit in slot 1 contains an *adder* and a *multiplier*, and the output of the execution unit is obtained by selecting the output of one of these two operators with a multiplexer. Then, a fault in the adder will not affect the behavior of the multiplier. Therefore, the fault state can be defined as $fsEU(1,adder) := 0$ and $fsEU(1,multiplier) := 1$. The fault state of a read port is given by the fault state function

$$fsRP : \text{RPS} \times \text{REGS} \rightarrow \{0,1\}.$$

Thereby $fsRP(p,r) = 1$, if and only if register r can be accessed correctly through read port p. The fault state of the bypass is given by two functions

$$fsBPex : SLOTS \times \text{BPS} \rightarrow \{0,1\} \text{ and}$$

$$fsBPwb : SLOTS \times \text{BPS} \rightarrow \{0,1\}.$$

Thereby $fsBPex(s,b) = 1$, if and only if a value is correct forwarded from the EX-stage of slot s through the bypass b. Similar, $fsBPwb(s,b) = 1$, if and only if the forwarding from the write-back stage of slot s through bypass b is working properly. Finally the state of the register file is defined by the function

$$fsRF : \text{REGS} \rightarrow \{0,1\}.$$

Here, $fsRF(r) = 1$, if and only if a value can be stored correctly in register r by an arbitrary slot. The fault state of the control path is given by the fault state function

$$fsCP : \rightarrow \{0,1\},$$

which is a constant function with arity 0. Thereby $fsCP = 1$, if and only if the control path is faultless. The composite fault state function

$$fsSlot : SLOTS \rightarrow \{0,1\}$$

reflects the fault state of the slots. Thereby, $fsSlot(k) = 1$ means that slot k can be used for executing some operations distinct from a *NOP*. $fsSlot(k) = 0$ means that only a *NOP* can be executed correctly in slot k. The fault state function *fsSlot* is derived from the previously specified fault state functions. How the function *fsSlot* is obtained from the other fault state functions depends on the *granularity level* for which the fault state of the VARP processor is specified.

Definition 2-1 (*Fault State of the VARP Processor*):
The *fault state* of the VARP processor is given by a set of fault state functions that reflect the fault state of particular components of the VARP processor.

The granularity levels are given in table 2-1. Their meaning is specified with the help of a fault-tree-like hierarchical representation of the VARP processor in figure 2-4. This representation considers the processor as the top-level system composed of the two sub-systems *control path* and *data path*. The data path is further divided into more sub-systems. Thereby sub-systems may be also represented by their provided functionality. For example, the read port contains sub-components that must be operational for reading a particular register trough this read port. The bypass has sub-components that must be operational in order to allow for selecting a value from a particular pipeline stage (*EX-stage* and *WB-stage*) or from

the read port (*register*). Inner nodes of the tree represent components whose functioning depends on the functioning of other sub-components, while a leaf represents an atomic component or its function. Thereby the fault state of a leaf is given by the annotated fault-state function. Similar to fault-trees, the fault state of inner nodes is derived from the fault states of their sons. Then, it can be determined bottom-up whether or not the processor is functional.

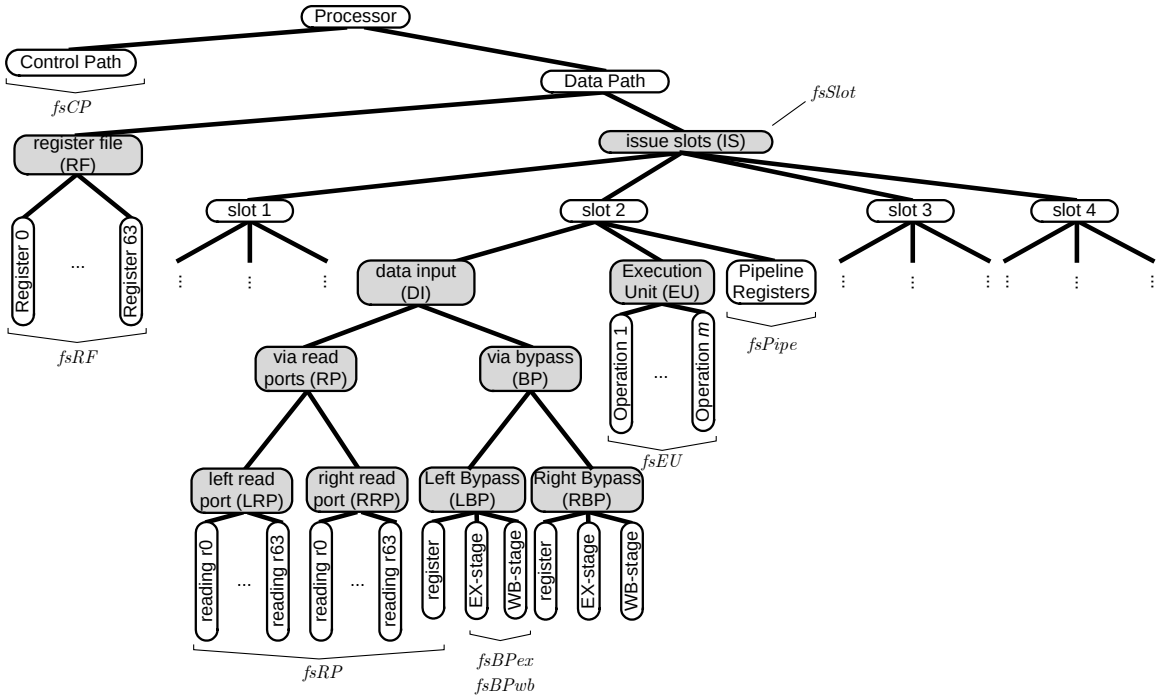


Figure 2-4: Hierarchical organization of the components of the VARP processor.

In the simplest case the fault state of an inner node is determined by a conjunction or disjunction of the fault states of the sons. For example, a non-fault tolerant VARP processor is only operational, when all its components are faultless. Thus, all atomic components must be operational (i.e., their fault state is 1), and the operational state of each inner node is determined by a conjunction of the operational states of its sub-components. I.e., all components of the processor are non-fault tolerant. The third column of table 2-1 shows for each granularity level the fault tolerant nodes of the tree. These fault tolerant nodes have redundantly available sub-components that can replace each other, if one of them becomes faulty. In most cases the fault state of a fault-tolerant node is determined by a disjunction of the fault states of its sons. I.e., it is sufficient that at least a single son is functioning. However, in some situations a particular subset of the sons must be functioning. The fault state of nodes not listed in the third column of table 2-1 is determined by a conjunction. Based on this information and the bottom-up evaluation, the fault state function *fsSlot* is derived from the other fault state functions. The fault state functions, which are sufficient for characterizing the fault state of the VARP processor at a particular granularity level, are listed in

the second column of table 2-1. Please note that the function *fsCP* is not listed there, because $fsCP = 1$ is mandatory for having an operational VARP processor.

Granularity Level	**Characterized by Fault State Functions**	**Fault tolerant nodes of the tree in** figure 2-4
Slot Level	*fsSlot*	*IS*
Execution Unit Level	*fsSlot*, *fsEU*	*IS, EU*
Read Port-Level	*fsSlot*, *fsEU*, *fsRP*	*IS, EU, RP, LRP, RRP*
Bypass-Level	*fsSlot*, *fsEU*, *fsRP*, *fsBPex*, *fsBPwb*	*IS, EU, RP, LRP, RRP*, *DI*, *BP*, *LBP*, *RBP*
Register-Level	*fsSlot*, *fsEU*, *fsRP*, *fsBPex*, *fsBPwb*, *fsRF*	*IS, EU, RP, LRP, RRP*, *DI*, *BP*, *LBP*, *RBP*, *RF*

Table 2-2: Granularity Levels for Self-Repair.

At *slot level*, only the node *IS* is considered as fault tolerant. Hence, each slot forms a redundantly available component, and only slots can replace each other. However, all sub-components of an operational slot must be faultless. I.e., a single fault in one of the sub-components of a slot implies that this slot is not operational anymore. Thus, a fault state at slot level is sufficiently characterized by the fault state function *fsSlot*. Whether or not the node *IS* is operational depends on the number of slots that are allowed to be faulty. If faults can be handled only in a single slot, then three sons of the node *IS* must be operational. If fault can be handled in up to three slots, then the fault state of the node *IS* is determined by a disjunction. The slot level is the most coarse-grained level for fault handling.

At *execution unit level*, a slot can be still operational when its execution unit has some faults. But at least a single operation distinct from a NOP must be executable in that execution unit, and all other sub-components of that slot must be faultless. For this reason, the operational state of the node *EU* in figure 2-4 (only shown for slot 2) is determined by a disjunction of the operational states of the operators in that execution unit. The fault state of the processor at that granularity level is characterized by the fault state functions *fsSlot* and *fsEU*. Both functions are needed, because a slot k may be operational, i.e., $fsSlot(k) = 1$, although there is an operator t in EU k, which is faulty, i.e., $fsEU(k,t) = 0$. Furthermore, a slot may be faulty, although all operators of the corresponding execution unit are faultless. Such a situation occurs for example, if a read port in that slot is faulty.

At *read port-level*, a slot is operational as long as some data from the register file can be provided to the execution unit by at least a single read port. However, at

read port level, the bypass must be faultless. For this reason, the operational state of the node data input (DI) is determined by a conjunction, while the operational state of the nodes RP, LRP, and RRP is determined by a disjunction. The fault state of the read ports is given by the fault state function *fsRP*.

At *bypass-level* also the bypasses are allowed to have faults. A slot is operational as long as some data can be provided to its execution unit; either from the register file or from other slots via bypasses. For example, a slot is considered as operational, even when both read ports and a single bypass of that slot are not operational. Then, at least a single operand for the execution unit can be provided. However, this may imply that operations with two operands cannot be executed in that slot. These operations must be specified as faulty by the fault state function *fsEU*.

The *register-level* additionally allows for tolerating faults in the registers of the register file. By this, the register file is operational as long as there is some operational register. Registers are considered as faulty, either because writing into them is not possible or the stored information is changed due to fault in the register. Please note that a fault in a register is equivalent to faults in all read ports when reading this register:

$$fsRF(r) = 0 \Leftrightarrow \forall k \in SLOTS : fsRP(2 \cdot k, r) = 0 \wedge fsRP(2 \cdot k + 1, r) = 0$$

The proposed granularity levels in table 2-1 are listed from coarse-grained to fine-grained. They are used throughout this thesis for specifying the granularity-level for fault handling by means of self-repair. The finer-grained the level, the smaller are the components that are allowed to be non-operational. As a consequence, a fault handling method for finer-grained levels allows for more components to be faulty without losing the functionality of the processor.

2.2.2 Programming and Simulation Model of the VARP Processor

Programming of the VARP processor is supported by a configurable assembler and an experimental compiler. In this thesis only the assembler has been used. Exploring the available instruction level parallelism of a program is the task of the compiler or the assembler programmer. Thus, already the assembler code is parallelized code that is just a more readable representation of the binary encoding of an instruction that is shown in figure 2-2. Formally each instruction w is a total function

$$w : SLOTS \rightarrow OPS$$

that maps each slot to an operation from the set *OPS*. This mapping is also called *binding*, because each operation is bound to a particular slot by instruction w. The set *OPS* is the set of all VARP-operations, and is given by

$$OPS := \mathcal{O} \cup \big(\mathcal{O} \times \mathbb{N}\big) \cup \big(\mathcal{O} \times \mathbb{N} \times \mathbb{N}\big) \cup \big(\mathcal{O} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N}\big).$$

This reflects the fact that each operation has a particular *type* $t \in \mathcal{O}$ and at most three other arguments, which are natural numbers. The type t is also denoted as *opcode*. For each operation type t there must be an *operator* of the same type in each execution unit of the VARP processor. Given an operation $m \in OPS$, the function $type(m)$ returns the type of m. The meaning of an argument of an operation m depends on the type of m. An argument x may be a register number or an immediate value. If x is a register, then $x \in$ REGS must hold. The notation rx is used to indicate that the argument x is a register. If x is an immediate value, then $0 \leq m < 2^{16}$ must hold. For example $(add, 4, 5, 6) \in OPS$ is a legal operation, representing an *add*-operation with source registers *r4*, *r5* and destination register *r6*. For convenience this operation is also written as *add r4*, *r5*, *r6*. An instruction w for the VARP processor is composed of four operations. For convenience this can be also written as $w = (a, b, c, d)$, which means that $w(0) = a$, $w(1) = b$, $w(2) = c$, and $w(3) = d$. In order to denote that an instruction w contains a particular operation a, the notation $a \in w$ is used, meaning there exists an $i \in SLOTS$, such that $w(i) = a$.

A program p for the VARP processor is now considered as a sequence of instructions, where $p(i)$ denotes the i-th instruction in this sequence, with $i \in \mathbb{N}$. A basic block is defined as follows:

Definition 2-2 (Basic Block):
Let p be a program. A *basic block* b in p is a sequence of instructions of maximal length, such that the only instruction of this sequence that is the target of a branch operation is $b(0)$, and the only instruction that may contain a branch operation is the last instruction in b.

In order to fully utilize the available parallelism of a VLIW processor sophisticated scheduling techniques are employed by compilers [59, 84, 111, 113]. These techniques are based on global code motion; i.e., operations are moved across basic block boundaries. For example, trace scheduling uses information about frequently used paths in the program in order to constitute traces from the basic blocks along these paths. During scheduling, a trace is treated similar to a simple basic block, but it may contain branch operations for leaving the trace from the middle. However, the instruction sequence obtained by trace scheduling or other scheduling techniques can be separated into ordinary basic blocks according to definition 2-2. These basic blocks constitute the base for some scheduling techniques used during the software-based self-repair. I.e., basic block information is retrieved either from the assembler code or from the binary code and is independent from the scheduling algorithm used by the compiler.

Simulating a piece of binary code is done either by using an instruction set simulator – then the simulation is instruction accurate – or by running the assembler program using a cycle accurate VHDL model of the VARP processor. In both cases, the model of the program memory is initialized with the binary code for simulation; see figure 2-5.

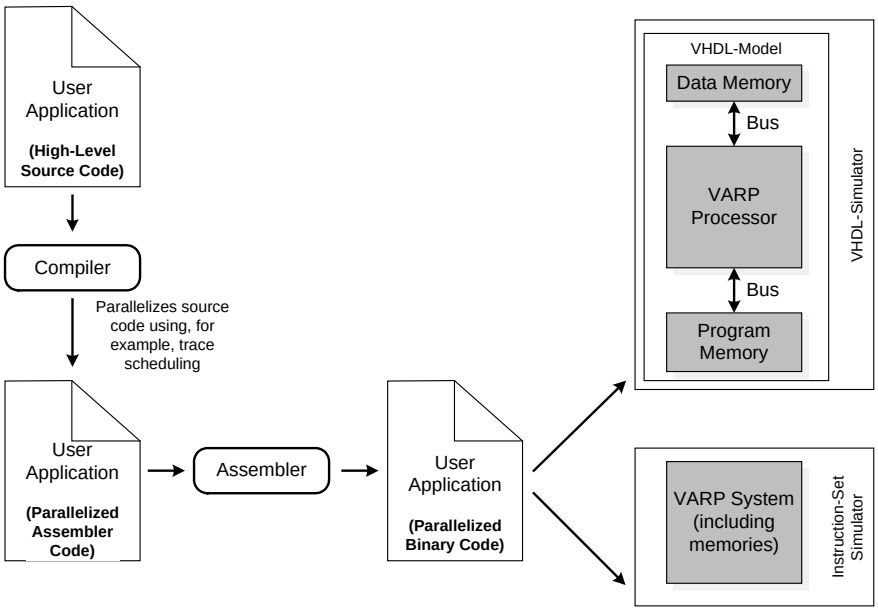


Figure 2-5: Tool chain and simulation environments for the VARP processor.

The VHDL model of the VARP processor is synthesizable. Synthesis results are reported in the subsequent section.

2.2.3 Reliability of the non-Fault Tolerant VARP Processor

The VHDL processor model has been synthesized using the Cadence RTL Compiler and the 45 nm *NangateOpenCellLibrary*[12] in order to obtain the cell area of each component of the VARP processor. Table 2-3 shows in a hierarchical manner the cell area and the number of instances of each component.

Component	Processor											
		Slot								Control Logic	Register File	
			Pipeline Registers				Read Port	Bypass	EU			Register
				FE	DE	WB						
Instances of components	1	4	1	1	1	1	2	2	1	1	1	64
Cell area per component	28811	4748	791	199	446	146	1153	240	1171	539	9280	145
Cell area relative to the processor area	100%	16,48%	2,75%	0,7%	1,5%	0,5%	4,00%	0,83%	4,07%	1,87%	32,20%	0,50%

Table 2-3: Cell area of various components of the 16-bit VARP processor in µm².

[12] http://www.nangate.com/ [last access: 14/01/20]

The full VARP processor requires a cell area of 28811 µm², which is the sum of the cell area of four slots, the register file, and the control logic. It can be noticed that the control logic (*Ctrl*) is very small (less than 2%) compared with the remaining data path of the processor. The size of a single slot is the sum of the sizes of two read ports, two bypasses, a single execution unit, and the pipeline registers. The size of the register file is the size of 64 registers, including the logic for writing these registers, but without read ports. Table 2-3 also shows in the last row the size of each single component in relation to the full processor area.

Based on these figures, the reliability of the non-fault tolerant VARP processor is computed, assuming a constant failure rate λ for a single cell area. Because all cells must be faultless in the non-fault tolerant VARP processor, the processor can be considered as a serial composition of all of its cell areas, yielding the reliability function

$$R_{NFT_i}(t) = \left(e^{-\lambda_i \cdot t}\right)^{28811} = e^{-28811 \cdot \lambda_i \cdot t} \tag{2-1}$$

that is plotted in figure 2-6 for various failure rates λ_i. In order to make the time scale of the reliability diagram meaningful, suppose that the failure rates were obtained by an accelerated life time test that was performed for 50.000.000 cells, each of them occupying a cell area of 1 µm². Furthermore, the accelerated simulation corresponds to 87.000 operational hours of the gates. I.e., a single time unit of the time axis in figure 2-6 represents 87.000 hours, which are approximately ten years. Then, for example, an observed constant failure rate of $\lambda_1 = 5 \cdot 10^{-7}$ means that running 50.000.000 cells for ten years, yields $50.000.000 \cdot \lambda_1 = 25$ faulty cells after ten years. Using this failure rate for a single cell area, the probability that a VARP processor is running after ten years is approximately 0.985. For R_{NFT4} with $\lambda_4 = 1 \cdot 10^{-5}$, this probability is only 0.75. A similar reliability function R_{real} is obtained by using the parameters proposed in [176], where a failure rate of $4 \cdot 10^{-6}$ for a processor is assumed using a time unit of one hour, which yields a MTTF of approximately 30 years. This shows that the used failure rates λ_1 to λ_4 can be considered as somewhat realistic.

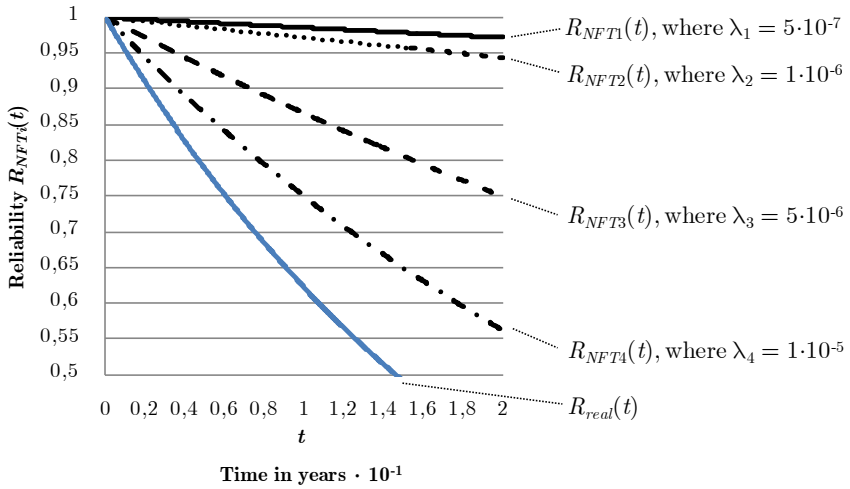


Figure 2-6: Plot of the reliability functions of the VARP processor for various failure rates, whereby $t = 1$ represents 10 years.

2.3 Hardware-based Rebinding in the VARP Processor

Now a simple hardware extension for the VARP processor is presented that allows for the reconfiguration of the VARP processor in the presence of single or multiple permanent faults in its slots. It is assumed that the fault state of the VLIW core is given at execution unit level by the fault state functions *fsEU* and *fsSlot*. Please recall that $fsEU(k,t) = 1$ means that operation of type t can be executed in slot k. The problem of detecting and locating permanent faults is discussed later in chapter 5.

2.3.1 The Rebinding Logic

The reconfiguration is based on the dynamic rebinding of operations to other slots – similar to the work of Chen in [43], where operations are dynamically scheduled into other slots. However, in contrast to the work of Chen, the rebinding of operations is performed by a simple *rebinding logic* that is added into the decode-stage of the VARP processor at the output of the fetch registers (see figure 2-7). First concepts for this rebinding logic were developed and implemented in [123].

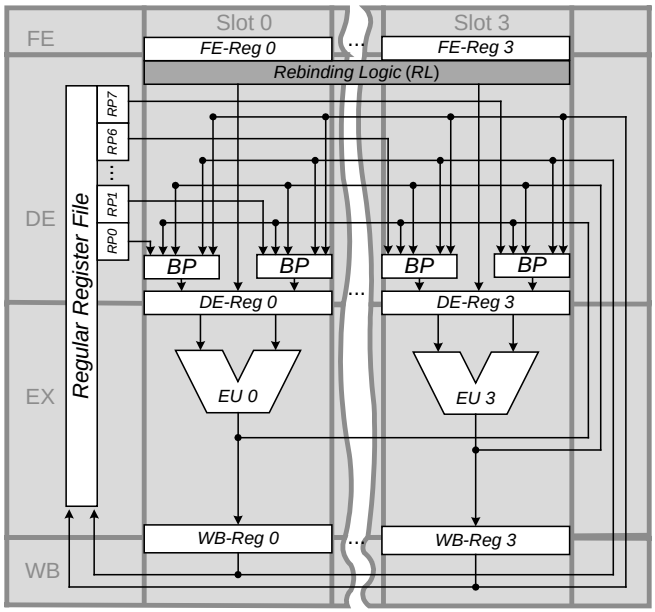


Figure 2-7: VARP-Core with rebinding logic.

Basically the rebinding logic is a multiplexer network, which is shown in more detail in figure 2-8. It can be considered as a simple dispatcher that can map the content of each fetch register or a NOP into any slot. Moreover, the fault state of the processor and the opcode, which is stored in each fetch register, are accessible for the *rebinding control logic*. The fault state of a slot is provided by a *fault state register* fsr_{slot}. Let $fsr(i)$ denote the i-th bit of a fault state register fsr, then $fsr_{\text{slot}}(k) = fsSlot(k)$ for all $k \in SLOTS$. In a similar way for each $k \in SLOTS$ there is a fault state register $fsr_{\text{eu}k}$ for which holds: $fsEU(k,t_i) = fsr_{\text{eu}k}(i)$, where $\mathcal{O}$ - {NOP} = $\{t_0,\ldots,t_m\}$ is the set of operation types supported by each execution unit. If the core is not faulty, then each multiplexer i in the rebinding logic selects the input from fetch register i. I.e., each operation is executed in that slot to which it was fetched, and all operations of an instruction are executed in parallel. No delay occurs during the execution.

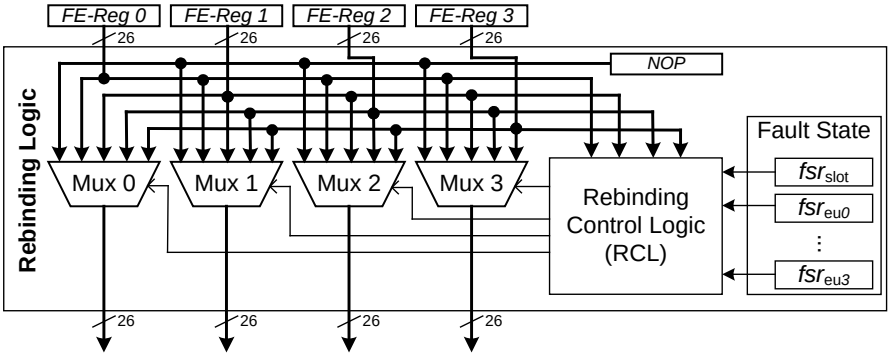


Figure 2-8: Details of the rebinding logic.

If $k \geq 1$ operations of the current instruction in the fetch registers cannot be executed in their originally assigned slot, then the rebinding control stalls the

fetching of a new instruction for k *rebinding cycles*. The current instruction is maintained in the fetch register and the multiplexer network is controlled in such a way that the non-executable operations are executed sequentially during the k rebinding cycles. Each of them is allocated dynamically to a slot that can execute this operation and it is replaced in the fetch register with a NOP. By this rebinding scheme the rebinding logic is kept simple, because it does not parallelize operations dynamically. Figure 2-9 illustrates the situation that two operations cannot be executed in the slot to which they were fetched. Only the pipeline registers of the VARP processor are sketched in figure 2-9. Suppose the four fetch registers contain the operations a, b, c and d. Furthermore, by the fault state functions it is determined that operation a cannot be executed in slot 0 and operation c cannot be executed in slot 2. In figure 2-9 (a) the first rebinding cycle is shown. Fetching of a new instruction is stalled, operation a is allocated to slot 1, and a is replaced in the fetch register with a NOP.

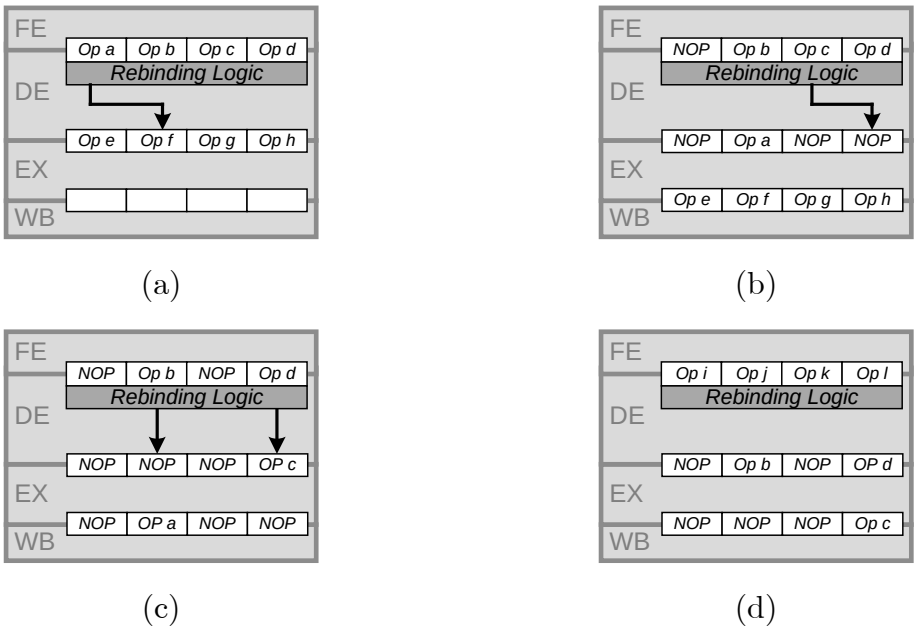


Figure 2-9: Example for fault handling by using the rebinding logic. (a) First rebinding cycle for operation *a*. (b) Second rebinding cycle for operation *c*. (c) Release cycle. (d) Normal operation continuous.

The values of the pipeline registers after the first rebinding cycle are shown in figure 2-9 (b). The second rebinding cycle is used for rebinding operation c. The result of the second rebinding cycle is shown in figure 2-9 (c). Each rebinding cycle inserts dynamically a new instruction into the decode stage. Each of these instructions contains exactly one of the non-executable operations from the fetch register, whereby this operation o has been allocated to a slot k with $fsEU(k,\text{type}(o)) = 1$ and $fsSlot(k) = 1$. Rebinding only a single operation per clock cycle simplifies the design of the rebinding control. After rebinding both operations the *release cycle* follows in figure 2-9 (c). During the release cycle the remaining

operations of the stalled instruction are executed. These operations can be executed in parallel, because they are executed in their originally assigned slots. Furthermore, during the release cycle the next instruction is fetched. Figure 2-9 (d) shows the values of the pipeline registers after finishing the release cycle. Please note that the release cycle may only execute NOPs. This is always the case, when all non-NOP operations of an instruction must be bounded to other slots.

2.3.2 Correctness of the Approach

Now it is shown that the rebinding cycles do not change the meaning of the executed program. For this reason this section will go into some details that were not mentioned above, and it will clarify some restrictions that must be respected in order to maintain the meaning of the program. Basically the rebinding logic may change the schedule of the program by executing operations sequentially that were scheduled in parallel. When changing the schedule, two types of dependencies in a program must be respected in order to maintain the meaning of the program [10]:

- Data dependencies and
- Control flow dependencies.

In the following it is shown that the sequential execution during rebinding cycles does not violate dependencies in the program.

2.3.2.1 Control Flow Dependencies

Control flow dependencies arise from conditional branches in the program. An instruction a is control flow dependent on a conditional branch in instruction b, if the execution of a depends on whether b takes the branch or not [10]. In particular, when the processor executes a branch in instruction b, the instruction a that follows instruction b must not be executed when the branch is taken, except b branches to a. In order to show that all control flow dependencies are respected, it is shown that the sequence of executed instructions is not affected by rebinding cycles. There are two types of control flow related operations in the instruction set of the processor:

1. Operations that write a value into the program counter (PC). These are branch operations that branch to a particular target address. They will be referred to as *Write PC-operations* (*WPC-operations*).
2. Operations that read the value of the PC. These operations do not immediately affect the control flow, but the value may be used by other operations for branching, e.g. for returning from a function call. These operations will be referred to as *Read PC-operations* (*RPC-operations*).

First, consider the situation that there is no control flow related operation in the pipeline or a branch operation that does not take the branch. Then a non-executable instruction in the fetch register will cause the rebinding logic to stall incrementing the PC until the release cycle is performed. Thus, the sequential fetching of the original instruction sequence is maintained.

Second, consider the situation that a rebinding cycle is performed, when a RPC-operation is in the pipeline. The behavior of the RPC-operation is not affected by rebinding cycles, because for each fetched instruction the corresponding value of the PC is buffered in the fetch register, too. When the instruction is moved to the decode register, then the buffered PC value is moved into the decode register, too, such that it can be used in the execution stage from the RPC-operation. Thus, the RPC-operation uses the same value as in the original instruction sequence.

Finally, consider the situation that a WPC-operation enters the pipeline. The normal execution of a WPC-operation is shown in the upper part of figure 2-10. A WPC-operation generates the new address a during execution stage (clock cycle t). This value is written to the PC at the end of the execution stage, such that the PC is equal to the new address a in clock cycle t+1. Moreover the reset signals for the fetch- and decode registers are generated in clock cycle t, such that the pipeline registers will have the values shown for clock cycle t+1 in the upper part of figure 2-10.

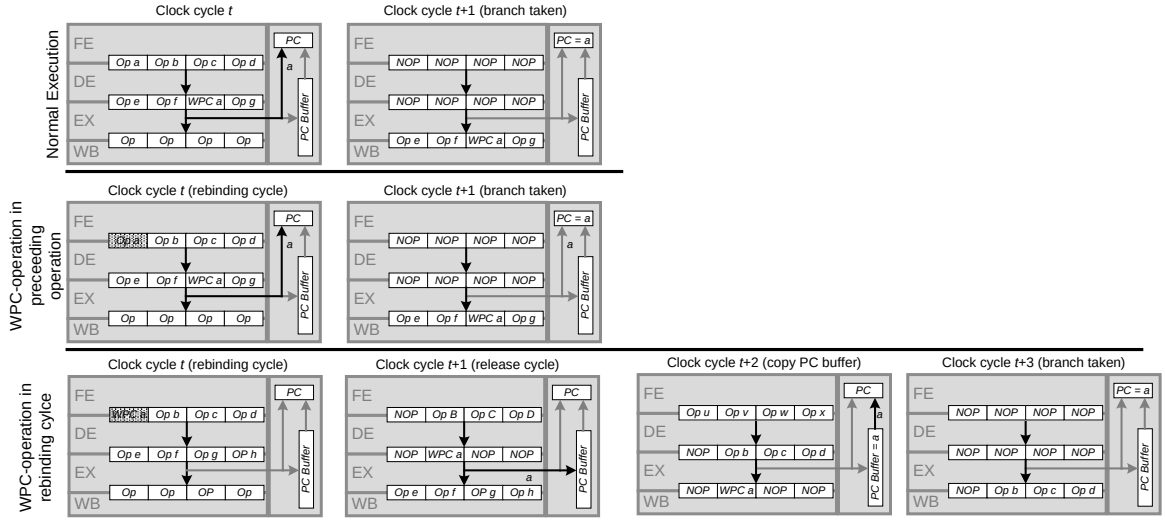


Figure 2-10: Buffering of the new PC address.

Both actions, generating reset signals and writing into the PC, must be delayed, when a WPC-operation is executed during a rebinding cycle. First consider the situation shown in the middle of figure 2-10, where a non-executable instruction w has been fetched (sketched by the shaded operation). This is the same situation as illustrated in figure 2-9 (a). Moreover, a WPC-operation from a preceding instruction w' is in the decode register. The WPC-operation enters the execution

stage, when the first rebinding cycle is performed for w during clock cycle t. Such a situation is not critical, because the WPC-operation in w' will reset the pipeline in the case that the branch is taken, and the non-executable instruction w has no effect. Thus, updating the PC and resetting the pipeline must be permitted for the first rebinding cycle.

Now consider the situation that the WPC-operation belongs to the non-executable instruction w in the fetch register. Two sub-cases exist. First, the WPC-operation is assigned to another slot during a rebinding cycle. Second, the WPC-operation remains in the fetch register until the release cycle is performed. The latter case is not critical, because instruction w will enter the execution stage after the release cycle. Then the branch operation is executed just like during normal execution and no further action is required. The first case is shown in the lower part of figure 2-10. The branch target address and the reset signal for the pipeline registers must be buffered when the WPC-operation enters the execution stage during a release or rebinding cycle. The buffered PC value and the reset signals are send to their designated receivers, one clock cycle after the release cycle. This is the point in time, when the PC would have been updated during normal execution of w. Please note that the instruction fetched in clock cycle t+2 is dropped from the fetch register due to the reset. Hence it does not matter whether this instruction is an executable one or not.

It can be summarized that WPC-operations executed during the first rebinding cycle are executed in the regular way. Generated values/signals of a WPC-operation that is executed during a rebinding cycle, except the first one, or during a release cycle are buffered. The first clock cycle after the release cycle is used for sending the buffered values/signals to their designated receivers.

2.3.2.2 Data Dependencies

Data dependencies arise from the access of two operations a and b to the same data location, where at least one of both operations must be a write-operation. Moreover, a and b must belong to distinct instructions. Data dependencies can be distinguished into three categories:

- *True-dependency*: b depends on a, if a writes to a location that is read by b.
- *Anti-dependency*: b depends on a, if a reads from a location that is written by b.
- *Output-dependency*: b depends on a, if a writes to a location that is written by b, too.

From the considerations in the previous section follows that the execution order of all instructions is maintained even when rebinding cycles are created. As a

consequence, data dependencies between different instructions will be maintained, too. This can be easily understood by looking at operation *a* in figure 2-11 (a). Each dependency from operation *a* to an operation in a successive instruction is maintained, because operation *a* and all other operations from cycle 1 are executed before an operation from a successive instruction is executed, no matter whether rebinding-cycles are introduced or not.

However, due to the rebinding cycles, new dependencies between the operations of a non-executable instruction may occur that were not present before, because the operations are executed in a sequential order. This situation is shown in figure 2-11 (b). There a feasible execution order for the operations from cycle 2 in figure 2-11 (a) is shown. Operation *add r3,r6,r7* is executed after operation *add r1,r2,r3*. This creates a new true-dependency between both operations, and the operation *add r3,r6,r7* would use a wrong value, because it reads the value stored by operation *add r1,r2,r3* in register *r3*. Figure 2-11 (c) shows another execution order of the operations from cycle 2 that generates a new anti-dependency. However, this anti-dependency is not critical, because the correct value is read from register *r3* before *r3* is redefined. Please note that new output dependencies will not occur. If a new output dependency would occur, then two operations within the same instruction must define the same destination register, which is not allowed.

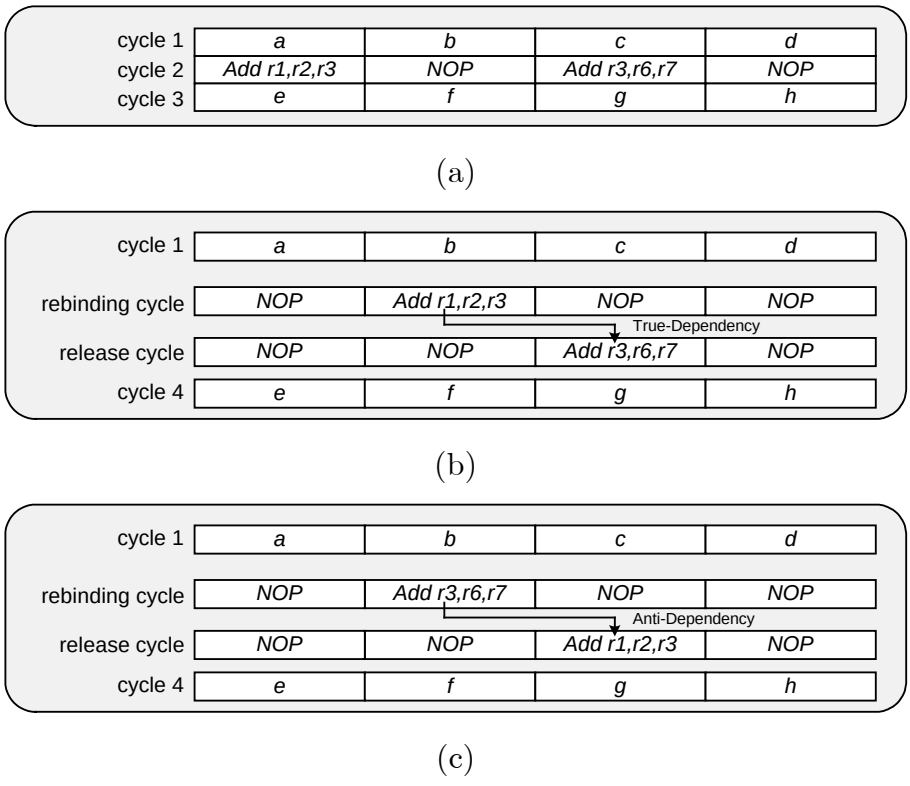


Figure 2-11: (a) Original instruction sequence. (b) Instruction sequence from (a) with rebinding cycles generating a new true-dependency. (c) Instruction sequence from (a) with rebinding cycles generating a new anti-dependency.

No hardware extension is included in the VARP processor for handling the situation shown in figure 2-11. Rather this problem is solved at software-level by following the paradigm of VLIW processors, where complex administrative tasks are shifted to the compiler. In order to avoid the occurrence of new true- and anti-dependencies, each instruction in the program must have the no-input-is-output property:

Definition 2-3 (*no-input-is-output* property):
An instruction w has the no-input-is-output property, if and only if for each pair u, w of operations with $u \in w$, $v \in w$ and $u \neq v$ holds: the destination operand of u is distinct from all source operands of v.

I.e., it is not allowed for an operation to write a register that is read by another operation within the same instruction[13]. This property must be guaranteed by the compiler or assembler programmer when generating the schedule for the user application. At least it should be stated that the sequential execution of operations due to rebinding cycles may cause values to be read from the register file instead of from the bypass. However, this is not a problem, because the source of an operand is determined dynamically.

2.3.2.3 Memory and Multi-Cycle Operations

Considering only dependencies is not fully sufficient, because the sequential execution of operations during rebinding cycles may change the timing of the communication of the core with its environment. In the VARP processor this may affect memory operations. Please recall that memory operations were divided into two consecutive single cycle operations (*st0*, *st1* and *ld0*, *ld1*). In both cases the rebinding cycles may change the timing during the execution of two single cycle operations. For example, the *ld1* operation may not be executed one clock cycle after the execution of the *ld0* operation. For this reason the single cycle implementations of all multi cycle operations of the processor core must be designed in such a way that the time between two operations, which are expected to be executed consecutively, can be prolonged without changing the behavior of the operations. For the memory interface of the VARP processor this is guaranteed in the following way:

- The load-operation is split into the *ld0*- and *ld1*-operation. The *ld0*-operation moves a memory address into the MAR-register during the execution stage. The memory guarantees that the expected value can be read from the memory output in the next clock cycle. However, the value is provided as

[13] Memory-operations must not be considered, because reading and writing to the same memory location by a single instruction is not possible.

long as the address in the MAR does not change. This means that the expected value can be also read some clock cycles later.

- The store-operation is split into the *st0*- and *st1*-operation. The *st0*-operation moves the address and data value into the MAR- and MBR-register, respectively, and enables the write signal for the memory. The *st1*-operation disables the write-signal. If both operations are not executed consecutively, the behavior does not change, except that the memory stores the expected value multiple times at the same memory address.

2.4 Results

The presented hardware-based reconfiguration performs an on-line rebinding of operations. But the method is only dedicated for handling permanent faults that were detected off-line, because no techniques for concurrent error detection and error handling are included, as it has been done for example in the work of Chen et al. [43]. Multiple permanent faults can be handled either at slot level or execution unit level. At execution unit level, the correct execution can be guaranteed as long as each operation type can be executed at least in one slot, which must not be the same slot for all operations. By this finer-grained fault handling, the on-line rebinding provides more flexibility than an off-line reconfiguration scheme, as it was proposed for example from Koal et al. in [92], where all operations of a defect slot will be allocated to the same spare slot. But this flexibility comes at the cost of performance degradation. The impact of the hardware-based rebinding on the runtime and on the reliability of the VARP processor is discussed in the following two sub-sections.

2.4.1 Performance Degradation

A particular fault state of the VARP processor is given by the fault state functions *fsSlot* and *fsEU*. In order to evaluate the impact of various fault states on the runtime of various user applications, fault states that affect the same number of slots are grouped into a single fault state class.

Definition 2-4 (*k*-slot-fault):
A k-slot-fault is a set of fault states, where exactly k slots, $0 \leq k \leq |SLOTS|$, are faulty, i.e., $|\{i \mid i \in SLOTS \text{ and } fsSlot(i) = 0\}| = k$.

For example, consider a 1-slot-fault. This fault state class represents four different fault states of the VARP processor, where each fault state corresponds to exactly one faulty slot. Similar, a k-operator-fault is defined.

Definition 2-5 (*k*-operator-fault):
A *k*-operator-fault is a set of fault states, where exactly $k \in \mathbb{N}$ operators are faulty, i.e., $|\{(i,t) \mid i \in SLOTS \text{ and } t \in \mathcal{O} \text{ and } fsEU(i,t) = 0\}| = k$, and the remaining components of the slots are faultless.

Please note that fault states belonging to the same *k*-operator-fault class may affect operators in the same execution unit as well as in different execution units.

If a *k*-slot-fault, with $0 \leq k < |SLOTS|$, is handled at slot level, then the proposed hardware-based rebinding may cause a worst case runtime of the application of up to

$$wcrt_{slot}(k) = (k+1) \cdot 100\% \tag{2-2}$$

This worst case only occurs, when in each instruction k operations must be allocated to another slot. Then k rebinding cycles and a single release cycle are needed for each fetched instruction, whereby the release cycle corresponds to the normal execution of the instruction.

When *k*-operator-faults, are handled at execution unit level, then the worst case runtime

$$wcrt_{eu}(k) = \begin{cases} (k+1) \cdot 100\%, \text{ if } k < \left|SLOTS\right| \\ (\left|SLOTS\right|+1) \cdot 100\%, \text{ otherwise} \end{cases} \tag{2-3}$$

is obtained. If k operators in distinct execution units are faulty, then at most k operations per instruction must be rebound, which requires k rebinding cycles plus a single release cycle. The worst case runtime is bounded by the number of slots, because at most $|SLOTS|$ operations per instruction can be rebound to another slot, which requires $|SLOTS|$ rebinding cycles plus a release cycle that will only execute a NOP-instruction.

For real applications both worst-case scenarios will rarely occur, because the runtime overhead strongly depends on the schedule of the application. In table 2-4 a brief charachterization of various benchmark applications is given. Most of them are single loop kernels of various filter algorithms (auto regression filter, discrete cosine transformation, fast fourier transformation, elliptic wave filter) taken from [101]. The operations of each benchmark program were classified into *mul*-, *add*- and *sub*-operations. Moreover, the number of operations in each benchmark program as well as the critical path lengths are shown. In [157] for each of these benchmark programs a schedule was generated that can be executed efficiently on the non-fault tolerant VARP processor. I.e., the lengths of these schedules were optimized, such that the average number of operations per instruction is high. By the average number of operations per instruction it can be noticed that a large amount of instructions contain four operations. It is obvious that such schedules

with many operations executed in parallel will cause a higher runtime overhead after rebinding than schedules with a lower average number of operations per instruction, as they can be found in more control flow dominated parts of an application.

Name	add	sub	mul	total operations	critical path length	schedule length	Average number of operations per instruction
ARF	12	0	16	28	8	8	3.5
DCT-DIF	17	12	12	41	7	11	3.7
DCT-DIT	24	12	12	48	7	14	3.4
DCT-LEE	17	12	20	49	9	14	3.5
EWF	26	0	8	34	14	14	2,4
FFT	8	17	13	38	4	10	3.8
SWIM1	12	8	6	26	4	8	3.5
SWIM2	6	3	6	15	5	5	3.0

Table 2-4: Characterization of benchmark programs used for performance evaluation.

Different fault states of the VARP processor usually cause different runtime overheads for the same benchmark. Because each k-slot-fault and each k-operator-fault represents various fault states, table 2-5 reports the best-case and worst-case runtime overhead obtained for all the fault states belonging to a particular k-slot-fault respectively k-operator-fault.

	***k*-slot-fault**						***k*-operator-fault**					
	***k*=1**		***k*=2**		***k*=3**		***k*=1**		***k*=2**		***k*=3**	
	best	**worst**	**best**	**worst**	**best**	**worst**	**best**	**worst**	**best**	**worst**	**best**	**worst**
ARF	163%	200%	250%	300%	350%	388%	100%	163%	125%	225%	163%	288%
DIF	182%	200%	273%	300%	373%	391%	100%	182%	100%	245%	100%	300%
FFT	190%	200%	280%	300%	380%	390%	100%	170%	100%	240%	100%	300%
EWF	129%	193%	171%	271%	250%	314%	100%	193%	100%	250%	107%	286%
DIT	157%	200%	243%	300%	343%	386%	100%	164%	100%	214%	107%	257%
LEE	179%	200%	257%	293%	350%	371%	100%	179%	100%	257%	100%	300%
Average:	**167%**	**199%**	**246%**	**294%**	**341%**	**373%**	**100%**	**175%**	**104%**	**239%**	**113%**	**289%**

Table 2-5: Best-case and worst-case runtimes for the benchmark programs from table 2-4.

In most cases the worst-case runtime of the presented benchmark programs for k-slot-faults is close to the theoretical worst-case $wcrt_{slot}(k)$. This is due to the fact that the benchmarks provide a high amount of instruction level parallelism, such that most slots have to execute an operation in each clock cycle. For the same reason, also for 1-operator-faults the worst-case runtimes in table 2-5 are close to the theoretical worst case in equation (2-3), because the schedules of these benchmarks have at least one slot that executes only operations of a single type. When the corresponding operator in that slot is faulty, then all of these operations must be allocated to another slot. It can be noticed that for $k > 1$ the worst-case

runtime for k-operator-faults will not adhere the theoretical worst-case so closely, because in most other slots a mix of at least two operation types is available.

As a consequence, the worst-case runtime can be reduced at the cost of increasing the best-case runtime, when the original schedule contains for all slots a similar operation mix. I.e., for each operation type (including the NOP) the numbers of operations of that type in each slot should be almost equal. Then for each k-slot-fault and each k-operator-fault the worst-case and best-case will be almost equal. This, for example, allows for a better prediction of the performance degradation of real time applications in the presence of faults.

2.4.2 Reliability Analysis

The proposed hardware-based rebinding is able to handle various combinations of multiple faults. When faults are handled at execution unit level, then the system remains operational for a given application as long as each operation of the application can be executed at least in one slot. This fact makes the reliability estimation very complex, because the number of operational system states becomes very large. When faults are handled at slot level, then the system is operational as long as there is at least a single faultless slot. The reliability function for the latter case also provides a lower bound for the reliability of the system, when faults are handled at execution unit level.

In order to determine the reliability function for the case that faults are handled at slot level, the components of the VARP processor are partitioned into non-fault tolerant components and fault tolerant components. When a non-fault tolerant component fails, then the whole processor fails. When a fault-tolerant component fails, then the system can continuous to provide its service, as long as the functionality of the failed component can be replaced by another component. The partitioning is shown in table 2-6 together with the cell area of each component.

Component	**Fault Tolerant VARP processor with hardware-based rebinding**												
		Fault Tolerant Part of a Slot							**Fetch Reg**	**Control Logic**	**Rebinding Logic**	**Register File**	
			Pipeline Registers			**Read Port**	**Bypass**	**EU**					**Reg**
				DE	**WB**								
Instances of components	1	4	1	1	1	2	2	1	4	1	1	1	64
Cell area per component	30281	4549	592	446	146	1153	240	1171	199	539	1470	9280	145
Reliability function	-	$R_{slot}(t) = e^{-4549 \cdot \lambda t}$	-	-	-	-	-	-	$R_{fr}(t) = e^{-199 \cdot \lambda t}$	$R_{ctrl}(t) = e^{-539 \cdot \lambda t}$	$R_{rl}(t) = e^{-1470 \cdot \lambda t}$	$R_{rf}(t) = e^{-9280 \cdot \lambda t}$	-

Table 2-6: Cell area in µm² and reliability function for each component of the fault tolerant VARP processor with hardware-based rebinding, assuming a constant failure rate λ.

The fault tolerant components are shaded gray. The table also shows the synthesis results for the rebinding logic, which makes up a portion of 4.8% of the full processor area. Assuming a constant failure rate λ for a single cell area, each single component can be considered as a serial system of all its cells, leading to the reliability functions shown in table 2-6 for these components.

The operational dependencies between the fault tolerant components and the non-fault tolerant components of the VARP processor are modeled by the RBD in figure 2-12.

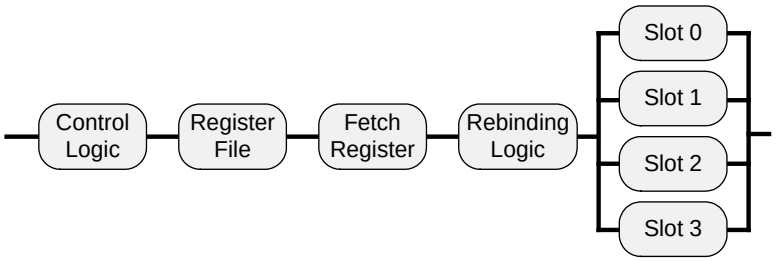


Figure 2-12: RBD for modeling the operational dependencies between components of the fault tolerant VARP processor.

The non-fault tolerant components from table 2-6 are composed into a serial system. The four slots form a parallel system, which means that at least one of them has to work properly. This yields the following reliability function $R_{hwr}(t)$ for the fault tolerant VARP processor:

$$\begin{aligned} R_{hwr}(t) &= R_{ctrl}(t) \cdot R_{rf}(t) \cdot R_{fr}(t) \cdot R_{rl}(t) \cdot (1 - (1 - R_{slot}(t))^4) \\ &= e^{-12085 \cdot \lambda t} \cdot (1 - (1 - e^{-4549 \cdot \lambda t})^4) \end{aligned} . \tag{2-4}$$

A plot of this reliability function for $\lambda = 1 \cdot 10^{-5}$ is shown in figure 2-13. There, it can be compared with the reliability functions plotted in figure 2-6 for the non-fault tolerant VARP processor. Comparing R_{hwr} with R_{NFT4} – both are plotted for the same failure rates $\lambda = \lambda_4$ – shows that the reliability of the fault tolerant VARP processor is improved. The reliability improvement factor RIF ranges in the plotted time interval from 2.3 to 2.0. Moreover, the fault tolerant system has a better reliability than the non-fault tolerant VARP processor with a failure rate of $\lambda_3 = 5 \cdot 10^{-6}$. This means, by applying hardware-based rebinding for fault handling at slot level, a more than two times bigger failure rate λ is acceptable to achieve the same reliability as in a system without hardware-based rebinding.

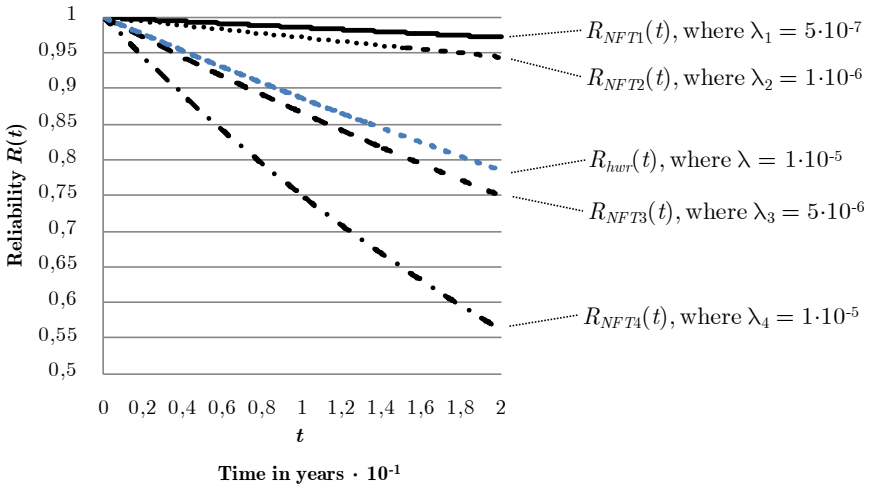


Figure 2-13: Reliability plots from figure 2-6 and of the fault tolerant core with hardware rebinding (R_{hwr}).

Another interesting observation can be made by comparing the reliability function $R_{hwr1}(t)$

$$R_{hwr1}(t) = e^{-30281\cdot\lambda t} + 4\cdot\left(e^{-12085\cdot\lambda t}\cdot e^{-4549{\cdot}3\cdot\lambda t}\cdot\left(1-e^{-4549\cdot\lambda t}\right)\right)$$

with the reliability function $R_{hwr}(t)$. R_{hwr1} is the reliability function for a fault tolerant VARP processor that can tolerate only 1-slot-faults. I.e., the processor is considered as operational as long as at most one slot is faulty. Both functions are plotted in figure 2-14.

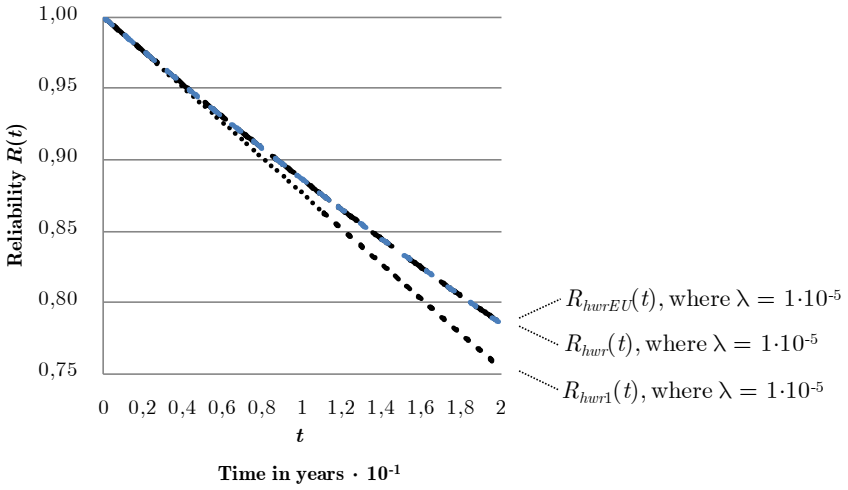


Figure 2-14: Reliability plot for fault handling at slot level when 3 slots are allowed to fail (R_{hwr}), 1 slot is allowed to fail (R_{hwr1}) and execution unit level (R_{hwrEU}).

The graphs of both function lie very close together, which means that the reliability of a VARP processor that can handle faults in up to 3 different slots is not much higher than the reliability of a VARP processor that can handle faults only in a single slot. In particular, when comparing R_{hwr} and R_{hwr1} the RIF ranges

for the plotted time interval between 1.0004 and 1.15. However, especially for those times, when $R_{hwr}(t) > 0.99$ and $R_{hwr1}(t) > 0.99$, the RIF is below 1.009. The reason for this situation is that the probability of having two or three faulty slots by such a low failure rate is very low. Thus, for low failure rates in many situations handling of faults in a single slot may be sufficient.

Furthermore, in figure 2-14 the reliability function for fault handling at execution unit level is plotted (R_{hwrEU}). This graph was determined in a numerical way by using the general combinatorial model presented in section 1.1.4.1. For the evaluation it was assumed that the execution units of the VARP processor are composed of two operators p and q of almost equal size. The VARP processor was considered as operational as long as at least one functioning operator p and one functioning operator q is available. Each of these operators must be in a slot where the remaining components (read ports, pipeline register, etc.) are functioning, but both operators can be in different slots. A C-program was used for computing all functioning system states and their probability of occurrence. The result is that the reliability function R_{hwrEU} is almost equal to the reliability function R_{hwr}. I.e., the reliability improvement achievable by fault handling at execution unit level is almost the same as the reliability improvement achievable with fault handling at slot level.

However, handling of faults in multiple components (e.g. multiple slots) becomes essential for higher failure rates. For a high failure rate, the probability increases that a single processor is affected by multiple faults. But, handling of multiple faults will be not effective, if the size of the non-fault tolerant components in the system is too big. In such a situation the probability is very high that one of the multiple faults affects one of the non-fault tolerant components. Hence, the system fails. For this reason figure 2-15 shows the reliability plot for a VARP-like system, in which only 2% of the processors cell area – which is approximately the size of the control logic – is assumed to be non-fault tolerant. For the remaining area of the processor it is assumed that it is organized in four slots that can replace each other until at least a single slot is functioning. The reliability is plotted for a failure rate of $\lambda = 1 \cdot 10^{-3}$, which is 100 time larger than the failure rate used for the plots in figure 2-14. Even higher failure rates are predicted, for example, in [49]. It can be noticed that fault handling in single slots (R_{hwr1}) increases the reliability very much compared with a non-fault tolerant system (R_{NFT}). Moreover, fault handling in up to three slots is for such a high failure rate much more beneficial than fault handling in a single slot only. This can be observed by comparing the reliability plot R_{hwr} (up to three slots are allowed to fail) with R_{hwr1} (only a single slot is allowed to fail). However, even for high failure rates a finer grained fault handling at execution unit level will not further improve the reliability. This can be observed by comparing the reliability plot R_{hwr} (faults are

handled at slot level) with R_{hwrEU} (faults are handled at execution unit level). Both graphs are almost equal.

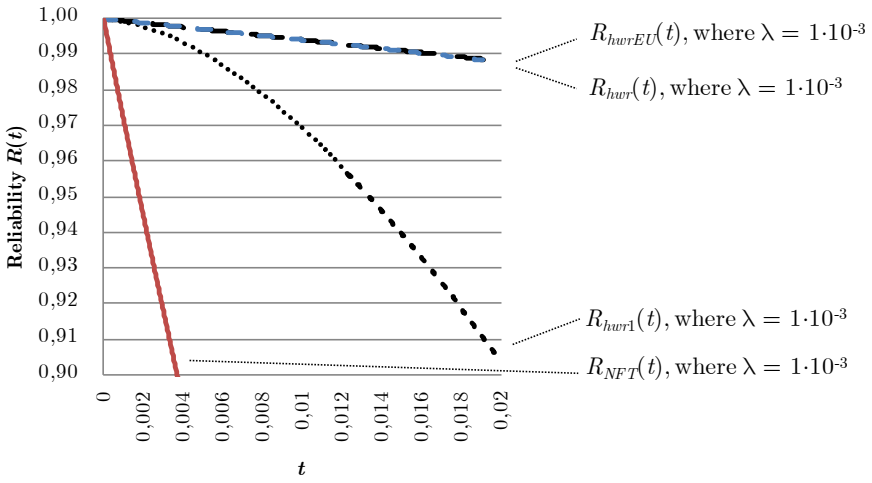


Figure 2-15: Reliability plots for very high failure rates in a VARP-like processor where only 2% of the processor area belongs to non-fault tolerant components.

2.5 Summary

In this chapter the non-fault tolerant VARP processor was introduced, which is used throughout this thesis for the demonstration of the proposed self-repair techniques. A simple model of that processor, which characterizes its key features, was introduced together with various fault state functions that are used for specifying the fault states of the VARP processor at different granularity levels. Techniques from literature for hardware-based administration of hardware redundancy were reviewed. Some basic concepts of these approaches were used for making the non-fault tolerant VARP processor fault tolerant by introducing a rebinding logic in the decode stage of the pipeline. This allows for handling multiple permanent faults at slot level and execution unit level during the runtime of the application by allocating operations dynamically to other slots. Nevertheless the approach is an off-line approach, because no concurrent error detection and fault localization is provided by the rebinding hardware. I.e., permanent faults must be detected and localized off-line. An appropriate test method is presented in chapter 5.

Synthesis results for the non-fault tolerant and the fault tolerant VARP processor were presented together with reliability plots for various failure rates. The reliability of the fault tolerant VARP processor is significantly improved in comparison with the non-fault tolerant VARP processor. However, it turned out that the reliability cannot be further improved significantly, neither by lowering the fault handling granularity from slot level to execution unit level, nor by

handling faults in multiple slots instead of handling them in a single slot only. This is due to the fact that the probability of having multiple faults in a processor is low, if the failure rate is low. But for high failure rates, fault handling in multiple slots will increase significantly the reliability of a VARP-like processor, compared with fault handling for a single slot only. However, this requires that the non-fault tolerant part of the processor is very small. By using hardware administrated hardware redundancy this cannot be guaranteed, because usually the administrative hardware components are error prone by themselves and belong to the non-fault tolerant part of the processor system. Even the proposed very simple rebinding logic consumes about 5% of the processor area. Moreover, the simple rebinding strategy increases the runtime overhead during the execution of the user application by almost 100% even for handling faults only in a single slot. The next chapter will show that with software-based methods both draw-backs, the additional hardware-overhead for administration logic and the runtime overhead, can be eliminated.

Chapter 3

Software-Based Self-Repair in a VARP Processor Environment

This section presents software-based approaches for handling permanent faults in the data path of a VARP processor by means of dynamic hardware redundancy [158, 161, 162]. Thereby the redundant hardware, which is provided by the superscalar data path of the VARP processor, is administrated by software such that the amount of additional hardware for administrating the hardware redundancy is reduced. First, a coarse grained approach is introduced in section 3.2 that works at slot- and operator-level. This approach does not need any hardware in the processor core itself for reconfiguring the data path. The reconfiguration is done in software by modifying the binary code of the user program in the program memory of the processor, such that the usage of faulty components in the data path is avoided during the execution of the user program. The modification of the program is carried out by a self-repair routine. It is shown that under particular conditions this self-repair routine can be executed by the faulty processor itself. The presented software-based solution in this section can be used to replace the hardware-based rebinding technique presented in chapter 2. In section 3.3 it is shown that the granularity level of the coarse grained approach can be refined without introducing too much complexity in the self-repair routine. Finally, the reliability improvement and performance degradation of the VARP processor implementing the software-based techniques is compared with the VARP processor that uses the hardware-based rebinding scheme from chapter 2.

3.1 Related Work

This section reviews software-based techniques for administrating hardware redundancy in processors. As shown in figure 1-9, the administration can be implemented at firmware-level or above. It is obvious that in multi-core systems the scheduler of the operating system can be used for avoiding the usage of faulty processor cores. Such a strategy is easy to implement, but it works at a very coarse-grained level. I.e., a single fault in a component of the core prevents the usage of the core. Firmware-level solutions were proposed that allow for fault handling at a finer-grained level. In [35] a software-based virtualization layer between the operating system and the hardware of the OPENSparc multi-core processor with eight cores is proposed. This virtualization layer is notified by the decode stage of a core, if a component that is needed by the currently decoded instruction is faulty. Then the virtualization layer is responsible for allocating and executing this instruction to another core. A similar idea is proposed in [87]. There a virtual machine is used that emulates instructions, whose needed execution resources are defect. Emulation is done in software with the help of other functioning resources. For example, floating point operations can be emulated with the help of integer units. However, in both approaches the problem of executing the virtualization software is not discussed. A very similar idea of emulating operations for single core microprocessors, but at firmware-level, was proposed by Benso et al. in [19, 20]. There, microprocessors are considered, whose data path is controlled by microcode. The usage of faulty data path components, when executing a particular assembler operation, is avoided by using another microcode sequence. However, statically scheduled superscalar processors like a VLIW usually do not use microcode. Moreover, changing the microcode sequence for a single operation may also change the timing of that operation, which disturbs the lock step execution with the other operations executed in parallel.

Not all abstraction layers shown in figure 1-9 must be available in an embedded system. For example, an operating system, a middleware layer or a firmware layer may be missing. Then hardware redundancy must be managed by the user application. Among others, this can be achieved by software implemented hardware fault tolerance (SIHFT), which is based on a duplication of code and data of the user application. The duplication can be done either at source code level or compiler-assisted. Furthermore, a correct control flow can be checked by introducing signatures for basic blocks, and checking whether or not the control flow enters a basic block from a legal control flow predecessor [9, 71, 109, 129]. In the work of Rebaudengo et al. code and data is duplicated at source code level [144]. This allows for the detection of temporary faults only, because duplicated operations may be executed by the same computational domain. Such an approach

must be classified as a kind of data- and time redundancy. However, when the duplication of operations is done by the compiler, as it is proposed by Bolchini et al. in [24] and other authors in [23, 82] for VLIW processors, then the compiler has control about the scheduling of operations, and the compiler can schedule the original operation and its duplicate into different slots. Then also permanent faults can be detected. However, SIHFT techniques, including the work of Bolchini, usually target for temporary faults only. Permanent faults are either not detected, because time redundancy is used as in the work of Rebaudengo, or permanent faults are detected, but the faulty component cannot be isolated from the system, as it is the case in the work of Bolchini.

In order to perform fault isolation, active hardware redundancy must be used. Some early work regarding software administrated active hardware redundancy for superscalar data paths was published by Guerra et al. [74, 75, 76] and Karri et al. [88, 89]. The targeted processor model is an application specific programmable processor (ASPP), which is simply an application specific integrated circuit (ASIC) that is partitioned into an application specific data path and a control path. The data path is tailored to a small set of applications by means of high level synthesis and has some similarities with the data path of a VLIW architecture. The data path of the ASPP provides parallelism at instruction level, and it is controlled by a statically scheduled program. For each application various fault tolerant schedules are generated during the development phase of the system. Each of them only utilizes a particular subset of the available data path components. Thereby permanent faults in the other data path components can be tolerated. If necessary, spare components are added to the data path, such that a priori specified set of faulty components can be tolerated. All pre-computed fault tolerant schedules must be stored in the program memory. Reconfiguration is done during the operational phase by selecting an appropriate schedule that does not use the currently faulty components of the ASPP. This approach is suitable for very small applications, and it becomes impractical for larger applications, because for many fault states of the processor, separate schedules are needed that become very long for large applications.

The L/U-reconfiguration scheme presented in [34, 130] for a similar ASIC architecture overcomes the problem of computing all schedules in advance. Only a single schedule is generated for a superscalar data path. This schedule is adapted during the operational phase to particular fault states, using the scheme shown in figure 3-1.

(a) (b)

Figure 3-1: Example of L/U-reconfiguration.

The original schedule is generated in such a way that it can be partitioned into a lower and upper part, as shown in figure 3-1 (a) by the bold line. Operations in the same row are executed in parallel, and operations in the same column are executed by the same computational domain. A computational domain corresponds to a slot in the VLIW architecture. If one of the computational domains fails then the upper and lower part of the original schedule are re-arranged as shown in figure 3-1 (b). By this, the required number of computational domains is reduced by one. This allows for fault handling at slot level, while the former mentioned work of Karri and Guerra allows also for fault handling at execution unit level.

A software administrated hardware redundancy scheme that is better suited for processor based systems that execute very large applications is presented by Meixner et al. in [118]. There a multi-processor system composed of simple RISC cores is considered, in which a master core is used for "detouring" a user application running on another core, when a fault is detected on that core. Detouring means that the user application is adapted in such a way that the usage of faulty components of the core is avoided. For this reason, the master core is running some kind of compiler that can compile the user application anew by taking into account the current fault state of the processor. Various detours are proposed. For example, some of them are based on redundancy available from bit-level parallelism. By this, a 32-bit addition can be divided into two 16-bit additions, and both 16-bit additions are carried out on a faulty 32-bit adder by using either the functioning lower or upper part of the adder. Other detours are proposed for faults in the register file and for faults in the bypass. Compared with the L/U-reconfiguration scheme, the detouring approach allows for a finer grained reconfiguration, but at the expense of a much more complex administration. In contrast to the on-line approaches that use a virtualization layer at firmware- or operating system layer, the detouring of user applications must be done in off-line mode. Details about the complexity of the compiler application running on the master core are not provided. Moreover, the proposed detours do not explicitly take advantage of the properties of statically scheduled superscalar processor architectures. This is done for the first time in the work of Schölzel [161] in order to reduce the complexity of the compiler-like application. In [161] various adaptation schemes for a user application running on a VLIW core are presented.

However, instead of a compiler-like program, a very simple self-repair routine is used. This routine only implements some simple adaptation techniques for the user application that can be performed easily in the field by the VLIW processor itself. Thus, in contrast to the detouring-approach, the self-repair approach can be used for single core systems, too. In the subsequent section a version of the self-repair approach for coarse grained fault handling is described in more detail. Furthermore, as already published in [162], it is shown that this granularity can be lowered without increasing the complexity of the self-repair routine too much.

3.2 Coarse-Grained Software-Based Self-Repair

The software-based self-repair approach is intended to be used in statically scheduled superscalar processors for handling permanent faults. As an example, it is applied to the non-fault tolerant VARP processor as shown in figure 2-1. I.e., a rebinding logic as it was added to the VARP processor in chapter 2 is not available. For this reason, the software-based self-repair performs this rebinding directly in the program memory of the VARP processor. This idea is illustrated in figure 3-2.

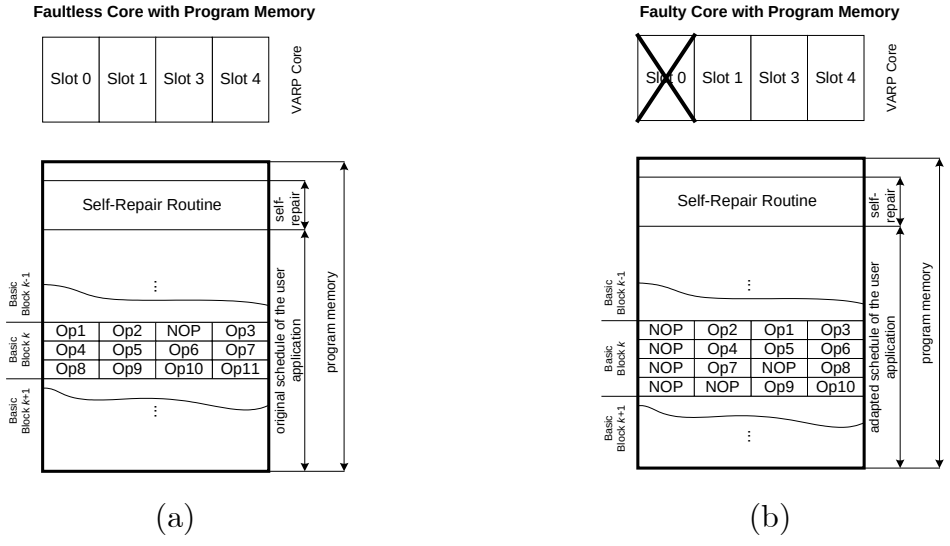


Figure 3-2: Concept of the software-based self-repair approach. (a) Faultless core with original user application in the program memory. (b) Faulty core with adapted user application in the program memory.

The compiler generated schedule of the user application is modified in the field by a simple self-repair routine. The self-repair routine is a program in the program memory that is used in off-line mode. I.e., when the user application is not running, a diagnostic self-test must be performed first in order to determine the fault state of the processor. The self-repair routine adapts the binary code of the user application to this fault state of the processor. Hence it is not necessary to

keep pre-computed schedules in the program memory. Rather the program in the program memory is modified permanently by the self-repair routine[14].

In figure 3-2 (a) a small fraction of the original compiler generated schedule for a faultless data path is shown. The data path has four issue slots and the schedule is divided into basic blocks according to definition 2-2. Now suppose that there is a fault in slot 0 of the VARP processor; i.e. $fsSlot(0) = 0$. In figure 3-2 (b) the adapted schedule of the user application is shown for this case. The schedule has been adapted by the self-repair routine in such a way that the defect slot is no longer used. I.e., all non-NOP operations from slot 0 were moved into other slots, and slot 0 executes only NOPs. Thereby, the assignment of operations to basic blocks is maintained. Please note that this kind of reconfiguration will only work, when the fault in slot 0 has a benign behavior. That is, the execution of a NOP operation in slot 0 does not affect the execution of operations in faultless slots[15]. Moreover, the length of the basic block k in figure 3-2 (b) is incremented by 1. This has various implications on the target address of branch operations. These implications are discussed in section 3.3.3. Now two alternative implementations of the self-repair routine are introduced. The first one is the *software-based rebinding* that will not increase the length of the basic blocks. With software-based rebinding, faults are handled at a coarse-grained level like slot- or execution unit level. The second implementation of the self-repair routine is the *software-based rescheduling* that may increase the length of some basic blocks, but it handles faults at finer grained levels like read port and bypass level.

3.2.1 Software-Based Rebinding

For each instruction w of the user application, the software-based rebinding computes a new binding w' for the operations in w. I.e., w' contains the same operations as w, but some of them are maybe allocated to different slots. By this, the length of the user application and each basic block remains unchanged. In order to perform the adaptation for the whole user application, instruction by instruction is transferred from the program memory into the data memory of the VARP processor. For each instruction a new binding is computed, and the

[14] Obviously it is required that the program memory is not a ROM. It is supposed that this restriction is not to strong because a rewriteable memory is used anyway; for example, for more flexibility when updating the software.

[15] An example of a malicious fault (non-benign fault) is a stuck-at fault that holds the write-enable signal for the destination register in slot 0 at 1. In this situation slot 0 writes a 0 in each clock cycle into register 0.

modified instruction is written back into the program memory. This process continues until all instructions of the user application are processed.

Whether or not an operation $w(k)$ from instruction w can be executed in slot k is determined by the fault state of the processor, given by the fault state functions *fsSlot* and *fsEU*. The rebinding algorithm will be explained in the subsequent sections only for fault handling at execution unit level. Fault handling at slot level is easily mapped to fault handling at execution unit level by considering all operators of an execution unit as faulty.

In the subsequent sections it is explained in detail how the instruction transfer between data- and program memory is accomplished, how the new binding is computed for an instruction w, and how the self-repair routine can be executed properly by a faulty VARP processor.

3.2.2 Transferring Instruction Words

The VARP processor is connected to the program and data memory as shown in figure 3-3 (a). There is an address bus *pmemAddr* from the core to the program memory and a data bus called *instruction* from the program memory to the core. In a similar way there is an address bus *dmemAddr* and a bidirectional data bus *dmemData* for connecting the core with the data memory. This bus structure causes three problems when transferring instructions from the program memory into the data memory of the original VARP processor:

- The VARP processor has no special instruction for accessing its program memory as data.
- Suppose there would be such an instruction. Then a resource hazard will occur, when the program memory is accessed by this data transfer operation.
- The bit width of an instruction word is different from the bit width of a data word.

In order to overcome these three problems, the transfer of instruction words from the program into the data memory and vice versa is organized by an *arbiter* that is integrated into the system as shown in figure 3-3 (b). The arbiter encapsulates the access of the processor to the data and address busses of both memories. By using the arbiter, a transfer of instructions between both memories can be done without a modification of the processor core.

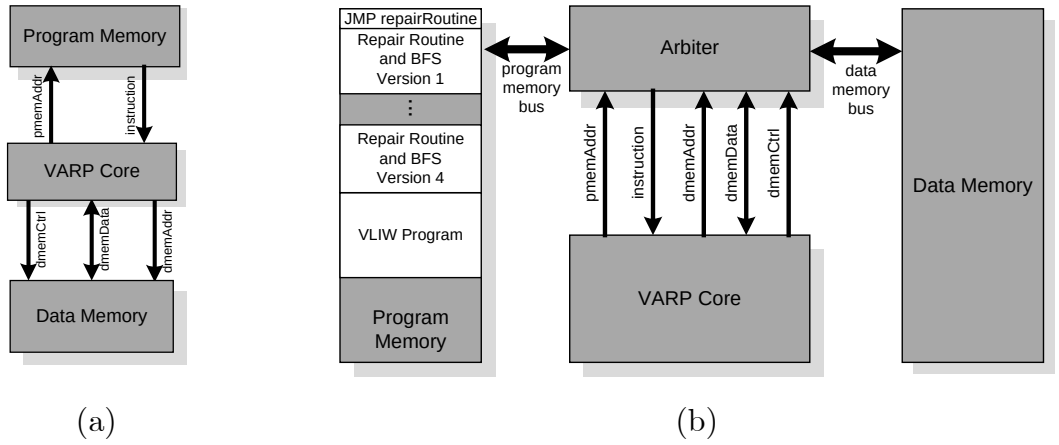


Figure 3-3: Memory bus system. (a) Original configuration. (b) Configuration with arbiter for instruction transfer.

The arbiter can run in three modes:

- **Normal-mode**: program and data memory can be accessed simultaneously by the VARP core. The arbiter simply forwards the signal *pmemAddr* to the program memory bus and the signals *dmemAddr*, *dmemCtrl*, and *dmemData* to the data memory bus. The instruction that is located in the program memory at address *pmemAddr* is forwarded to the core.
- **Read-mode**: The arbiter transfers an instruction from the program memory to the data memory. In this mode the VLIW-core can neither access the program nor the data memory. For this reason, the *instruction*-bus permanently delivers NOP-instructions to the core, no matter which instruction is located in the program memory at the current address *pmemAddr*.
- **Write-mode**: The arbiter transfers an instruction from the data memory to the program memory. The rest of the behaviour is the same as in the read-mode.

The VARP core must be able to initiate an instruction transfer from the program memory to the data memory and vice versa. In order to do so, a simple protocol is used for the communication between the core and the arbiter that can be implemented with the available *load*- and *store*-operations of the VARP processor. With this approach no modification in the core itself is needed.

When the arbiter is in normal mode, then it listens to the data transfer on the bus *dmemAddr*. If the arbiter detects a read- or a write-access to a special data memory address (for example 0xFFFF), then it switches either into the read- or into the write-mode. It switches into the read-mode, when the processor tries to read from the address 0xFFFF, and it switches into write-mode, if the processor tries to write to address 0xFFFF. The complete behaviour is shown in the state chart in figure 3-4.

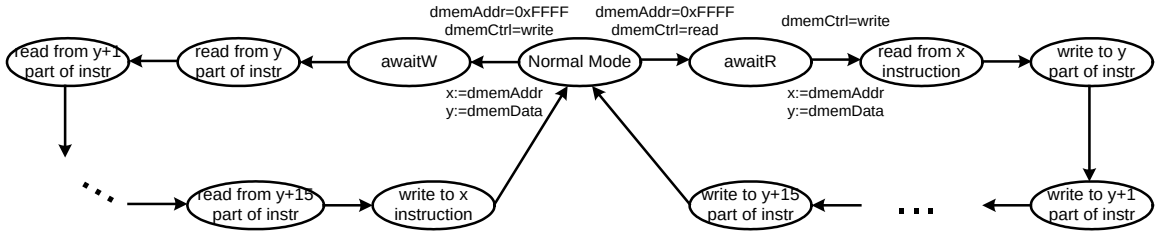


Figure 3-4: State chart of the arbiter.

When the access to address 0xFFFF was detected by the arbiter, then it is either in the *awaitW*- or in the *awaitR*-state, and it awaits a memory write operation of the core by listening on the busses *dmemAddr* and *dmemData*. Suppose that the core writes value x to address y. Then this write operation is not forwarded by the arbiter to the data memory. Rather, x is used by the arbiter as the program memory address from/to which an instruction must be read/written. Respectively, y is used as data memory address from/to which an instruction must be read/written. When the arbiter has received both values x and y, then it initiates the instruction transfer. The instruction transfer takes 17 additional clock cycles.

In read mode the instruction from the program memory address x is read by the arbiter. Each operation in that instruction is partitioned into four parts; each part has 16 bit. These four parts contain the opcode and the register numbers encoded in each operation. They are written for all four operations of an instruction to data memory addresses y to y+15. There they can be accessed easily by the subsequent *rebinding* routine. When the arbiter is in write mode, then, in a similar way, the instruction parts located at data memory addresses y to y+15 are composed to an instruction word, which is then written back to address x in the program memory. The partitioning of an instruction becomes necessary, because each instruction of the VARP processor is 104 bits wide, but the data memory has a bit width of 16 bits only. An example of this partitioning is shown in figure 3-5 for the instruction transfer from program memory address 0x10 to data memory address 0x56.

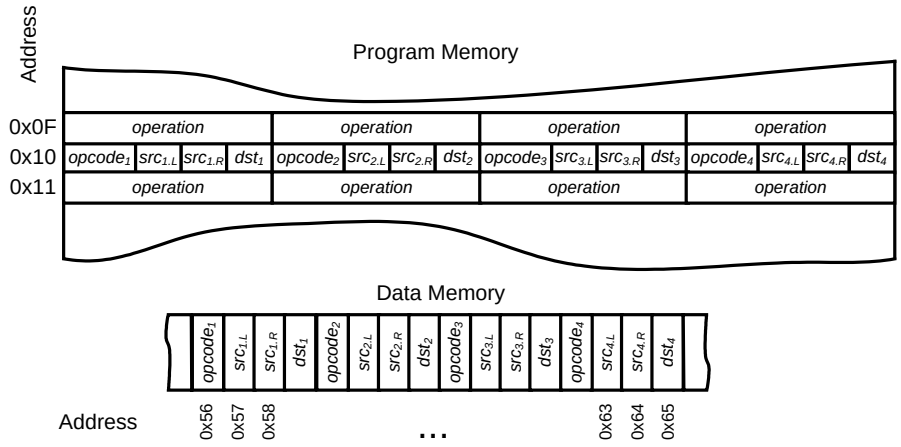


Figure 3-5: Example of an instruction transfer and the alignment of the 104 bit instruction word in the 16-bit data memory

How to implement the simple protocol for transferring instructions from the program into the data memory and vice versa is shown in the instruction sequences in figure 3-6. There an instruction transfer from the program memory address $x = 0x10$ to the data memory address $y = 0x56$ and vice versa is initiated.

```
ldc 0xFFFF, R0
ldc 0x0056, R2
ldc 0x0010, R3
ld0 [R0],R0    // switch into read mode
ld1 [R0],R0
st0 R3,[R2]    // submit addresses
st1 R3,[R2]
nop            // read instruction
nop            // write to y
nop            // write to y+1
nop            // write to y+2
nop            // ...
nop
nop
nop
nop
nop
nop
nop
nop
nop
nop
nop
nop            // write to y+15
ldc 0x1000, r9 // continuous normal
               // operation
```

(a)

```
ldc 0xFFFF, R0
ldc 0x0056, R2
ldc 0x0010, R3
st0 R0,[R0]    // switch into write mode
st1 R0,[R0]
st0 R3,[R2]    // submit addresses
st1 R3,[R2]
nop            // read from y
nop            // read from y+1
nop            // write to y+2
nop            // ...
nop
nop
nop
nop
nop
nop
nop
nop
nop
nop
nop
nop            // read from y+15
nop            // write instruction
ldc 0x1000, r9 // continuous normal
               // operation
```

(b)

Figure 3-6: (a) VARP-code for moving an instruction from program memory address 0x10 to data memory address 0x56. (b) VARP-code for moving an instruction from data memory address 0x56 to program memory address 0x10.

The first load-operation in figure 3-6 (a) accesses the data memory address 0xFFFF. This switches the arbiter into the *awaitR*-state. The next store-operation tells the arbiter that the instruction should be read from address 0x0010 in the program memory and stored in the data memory starting at addresses 0x0056. When the arbiter has received this information, the transfer starts. The transfer takes 17 clock cycles, because 16 data words must be stored in the data memory and one clock cycle is needed for reading the instruction. Note that the VARP core cannot read instructions or data while the transfer of an instruction is in progress. In order to avoid stalling the VARP processor, which may require modifications of the core, the arbiter delivers NOP-instructions to the VARP processor as long as the transfer is in progress. In order to make sure that the NOPs that the VLIW receives from the arbiter will really match with the instructions in the program memory, the listing in figure 3-6 (a) contains 17 NOPs after the initiation of the instruction transfer. By this the VARP processor behaves as if it would read these NOPs from the program memory although the program memory is not accessed by the arbiter. The assembler code in figure 3-6 (b) has the same structure, but it initiates the write mode of the arbiter. Note that both code examples only show

the code for a single slot of the VARP processor. The remaining slots execute NOPs.

3.2.3 The Rebinding Algorithm

Once an instruction w was moved from the program memory into the data memory, it can be modified there. This is accomplished by the rebinding algorithm. The goal of this algorithm is to find an instruction w', which is a permutation of the operations from instruction w, and each operation in w' is executed by an execution unit of the VARP processor that works properly on that operation. For example, consider a VARP processor with five slots and an execution unit configuration as it is shown in figure 3-7 (a). The instruction in figure 3-7 (b) cannot be executed properly in that data path, when the multiplier in execution unit 4 is faulty. But the instruction in figure 3-7 (c) with permutated operations can be executed properly. Thus, by computing such a permutation, the usage of a faulty component in the data path is avoided. Note that in this example the VARP processor does not have homogeneous execution units. This may reflect a situation in which already some operators of a homogeneous data path are faulty.

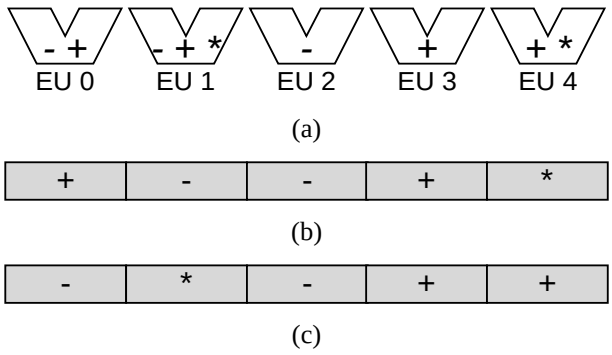


Figure 3-7: (a) Data path of a five issue VARP processor. (b) Original Instruction. (c) Instruction with permutated operations.

Definition 3-1 (Legal Permutation):
Given an instruction w, then a *legal permutation* of w is an instruction w' with the following properties: First, w' contains the same operations as w. Second, for all $k \in SLOTS$: $fsEU(k,type(w'(k))) = 1$ and $fsSlot(k) = 1$; i.e., operation $w'(k)$ can be executed correctly in slot k.

By constructing this legal permutation some of the operations in w may be allocated to other slots of the VARP processor. The computation of a legal permutation can be done efficiently. In order to show this, it is assumed that at most one operation per instruction cannot be executed by its assigned execution

unit[16]. When such an operation exists in w, then the corresponding execution unit is denoted as f (faulty). The problem of finding a legal permutation for instruction w and a given fault state can be modeled by a rebinding graph that is derived from w. A *rebinding graph* R is a directed graph (N,E), where $N = EUS$ is a set of nodes and $E \subseteq N \times N$ is a set of edges. The nodes represent the execution units in the data path. An edge (u,v) represents the possibility that operation $w(u)$, which is executed on execution unit u, can be executed on another execution unit v, too; i.e. $fsEU(v, type(w(u)) = 1$ and $fsSlot(v) = 1$. Thus, there is an edge (u,v) from node u to node v, if and only if:

- execution unit u executes an operation of type t in instruction w and
- execution unit v has an operator of type t and
- the operator of type t in execution unit v is not faulty, i.e. $fsEU(v,t) = 1$ and $fsSlot(v) = 1$, and
- $u \neq v$.

Consider as an example the rebinding graph in figure 3-8 that is constructed for the data path configuration from figure 3-7 (a) and the instruction from figure 3-7 (b). It is assumed that the multiplier in EU 4 is faulty. Thus there is no edge from node 1 to node 4.

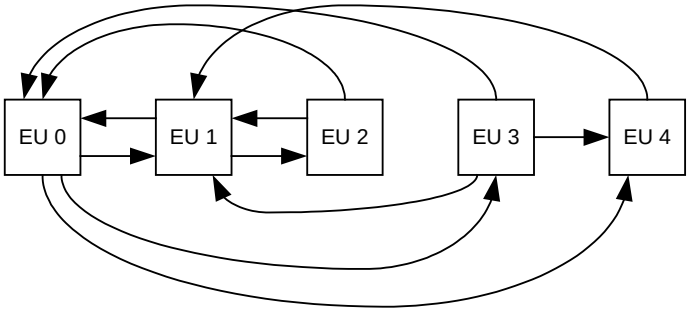


Figure 3-8: Example of a rebinding-graph.

But the multiplication that is executed on EU 4 can be executed on EU 1, too. Thus, there is an edge (4,1). The subtraction that is executed on EU 1 can be executed on EU 0 and EU 2, too. Hence, there are the edges (1,0) and (1,2) in the rebinding graph.

In order to compute the permutation, let $G \subseteq N$ be a set of functional units that contains only the faulty EU f and all EUs that execute a NOP in the given instruction w. G is called the *goal*. The problem is solved by finding a rebinding

[16] This restriction is relaxed later on. However, it is allowed that multiple operators of the EU are faulty.

path $v_0,v_1,\ldots,v_n$ in the rebinding graph, with the source node $v_0 = f$ and a sink node v_n that belongs to the set G.

Definition 3-2 (Rebinding Path):
Consider a rebinding graph (N,E) constructed for instruction w and fault states *fsSlot* and *fsEU*. A sequence of nodes $v_0,v_1,\ldots,v_n \in N$ is a rebinding path, if $n > 0$ and $v_0 = f$, $v_n \in G$ and $(v_i,v_{i+1}) \in E$ for all $0 \leq i < n$, and each node in the path is unique, except for the case $v_n = f$.

The permutation w' is obtained by shifting each operation that is executed by an EU on that path along one edge on that path to the successive EU. Hence, the permutated instruction w' is obtained by:

- $w'(v_{i+1}) \coloneqq w(v_i)$, for all $i \in \mathbb{N}$ and $0 \leq i < n$,
- $w'(v_0) \coloneqq w(v_n)$, if $v_n \neq f$ and
- $w'(k) \coloneqq w(k)$, for all $k \in N$ and $k \notin \{v_0,v_1,\ldots,v_n\}$.

As an example, consider again the rebinding graph in figure 3-8. There $G = \{4\}$, because the instruction in figure 3-7 (b) does not contain a NOP operation. A path that starts at source node 4 and ends at a sink node in G is 4, 1, 0, 4. Now each operation from the instruction in figure 3-7 (b) must be shifted along one edge on that path. I.e., operation * is shifted from EU 4 to EU 1, operation – is shifted from EU 1 to EU 0 and operation + is shifted from EU 0 to EU 4. By doing this, the permutation that is already shown in figure 3-7 (c) is obtained.

Theorem 3-1

Given an instruction w that contains only one operation that cannot be executed by their corresponding EU and a rebinding graph constructed for w. Then, shifting operations along an existing rebinding path in the rebinding graph computes a legal permutation.

Proof:

It is provided by the construction of the rebinding graph that an operation that is shifted along one edge on the path can be executed on the EU to which it is shifted. The first node on the rebinding path is the faulty execution unit. From that execution unit the only non-executable operation in instruction w is shifted away. Furthermore, the sink node of the path is either an EU that executes a NOP or the first node of the rebinding path from which the executed operation has been shifted away. In both cases the sink node is free for executing the operation that is shifted on it.

◊

Theorem 3-2

For each legal permutation w' of instruction w a rebinding path in the rebinding graph for w exists, such that w' is obtained by shifting the operations along that path.

Proof:

Let f be the faulty EU that cannot execute operation $w(f)$. Consider the permutation w' of w and let $\{v_0,v_1,\ldots,v_n\}$ be the set of EUs for which holds $w'(v_i) \neq w(v_i)$. That is, these EUs execute in w another operation than in w'. Because w' is a legal permutation, $w'(f)$ cannot execute operation $w(f)$. Thus, it must hold $f \in \{v_0,v_1,\ldots,v_n\}$. Without loss of generality let $v_0 = f$. Moreover, let $x_0 = w(f)$, $x_1 = w(v_1)$, $x_2 = w(v_2)$, …, $x_n = w(v_n)$ be the operations executed by the execution units $v_0,\ldots,v_n$ in instruction w. Furthermore, let $u_0,\ldots,u_n$ be the execution units for which holds $x_0 = w'(u_0)$, $x_1 = w'(u_1)$, …, $x_n = w'(u_n)$. Then operation x_0 is executed in w' by EU u_0, operation x_1 is executed in w' by EU u_1, and so on. This means that x_0 was shifted from EU v_0 to EU u_0, operation x_1 was shifted from v_1 to u_1, and so on. Thus $v_0,v_1,\ldots,v_n$ is the path of interest.

◊

From theorem 3-1 and theorem 3-2 follows that there exists a legal permutation w' of w, if and only if there is a rebinding path in the rebinding graph for w with source node f and a sink node from G.

In order to avoid the computation of a rebinding graph for each instruction of the user application, a *super rebinding graph* is now introduced that accumulates all possible rebinding graphs, and allows for a faster computation of a permutation without the construction of a particular rebinding graph for each instruction. A super rebinding graph $S = (N,E)$ is given by a set of nodes $N = EUS$, where each node represents an execution unit of the data path and a set of directed edges $E \subseteq N \times \mathcal{O} \times N$ that are labeled with operations. An edge (u, o, v) from source node u to destination node v is labeled with operation o if and only if, both execution units u and v can execute an operation of type o. This super rebinding graph is constructed only once for the faultless processor architecture. An example of such a super rebinding graph for the data path configuration in figure 3-7 (a) is shown in figure 3-9 (a).

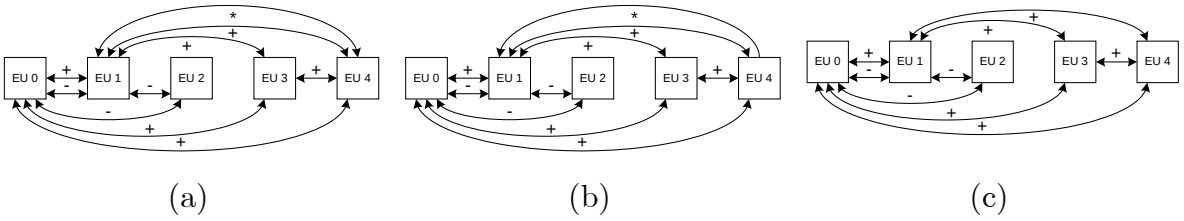


Figure 3-9: (a) Super rebinding graph for the data path configuration in figure 3-7 (a). (b) Intermediate super rebinding graph from (a) for the situation that the multiplier in EU 4 is defective. (c) Super rebinding graph for the degraded data path in which the multiplication in EU 4 is faulty.

In order to use the super rebinding graph for computing the permutation of an instruction, it must be adapted to the fault state of the processor given by *fsEU* and *fsSlot*. For simplicity it is assumed again that there is only a single faulty execution unit. Again, the faulty unit is denoted by *f*. If operators $\{o_1,\ldots,o_m\}$ in *f* are faulty, then all ingoing edges of node *f* that are labeled with $o_1,\ldots,o_m$ must be deleted from the super rebinding graph *S*. By this modification it is avoided that an operation of type $o_1,\ldots,o_m$ is moved along such an edge to node *f*. This yields an *intermediate super rebinding graph* S' that is used for computing the permutation of all instructions of the user application according to the current fault state of the processor. An example of such an intermediate super rebinding graph is shown in figure 3-9 (b). There the ingoing edges of node 4 labeled with a multiplication were removed, due to a fault in the multiplier of EU 4.

The rebinding graph R that is constructed for a given instruction w is a sub-graph of such an intermediate super rebinding graph S'. This can be easily validated for the nodes of R, which are isomorphic to the nodes of S'. Moreover, if there is an edge (u,v) in R, then there is an operation $p = w(u)$ in w with a particular type $o = type(p)$ that is executed by execution unit u. Moreover, execution unit v is able to execute p, too. Thus, in S' there is also an edge (u, o, v).

Moreover, each outgoing edge of node u in S' that is labeled with o is also found in R. Thus, the rebinding graph R is easily obtained from S' by deleting for each node n all outgoing edges that are not labeled with $type(w(n))$. For this reason, the problem of finding a path in the rebinding graph R can be also solved in the intermediate super rebinding graph S' by considering only those edges of S' that can be also found in R. For each node n of S', these are exactly those outgoing edges of a node n that are labeled with $type(w(n))$. Please note that it is not necessary to remove all the other edges from S'. For this reason S' can be used for computing the permutation of all instructions without being reconstructed for each instruction.

In order to find a rebinding path for a given instruction w in the intermediate super rebinding graph S' a breadth-first-search (*bfs*) is performed. The pseudo code for the *bfs* is given in listing 3-1.

```
1   function bfs(w, f, S')
2    foreach i ∈ N do
3      level[i] ≔ undefined;
4    od
5    from[f] ≔ undefined
6    currLevel ≔ 0;
7    level[f] ≔ currLevel;
8    do
9      foreach u ∈ EUS do
10       if level[u] = currLevel then
11         foreach v ∈ EUS do
12           if (u,type(w(u)),v) ∈ E then
13             if level[v] = undefined then
14               level[v] ≔ currLevel + 1;
15               from[v] ≔ u;
16             fi
17             if type(w(v)) = NOP or f = v then
18               return (v,from);                    // success
19             fi
20           fi
21         od
22       fi
23     od
24   while(∃u ∈ EUS with level[u] = k+1)
25 return (failed,from)                              // no success
```

Listing 3-1: Breadth-first-search algorithm *bfs* that searches for a rebinding path.

The function in listing 3-1 receives an instruction w, the faulty execution unit f, and the intermediate super rebinding graph S'. The *bfs* starts at the faulty node f, which gets the level 0. For all other nodes the level is *undefined*. Having the nodes with level k, a node v gets the level k+1, if and only if

- there is a node u in S' with level k and
- v has no level assigned and
- there is an edge (u, o, v) in S' and
- execution unit u executes an operation of type o in instruction w, i.e., $type(w(u)) = o$.

For each of the nodes v that is assigned to level k+1, the node u, from which v has been reached, is reminded by setting $from[v] := u$ in line 15 of listing 3-1. The *bfs* stops, if a node is reached that belongs to the goal G or no node can be assigned to level k+1. In the latter case no legal permutation exists, and, in the former case, a path can be constructed with the reminder array *from*. An example of the application of the *bfs*-search to the intermediate super rebinding graph from figure 3-9 (b) is shown in figure 3-10. There, the values $level[n]$ and $from[n]$ are annotated in each node n.

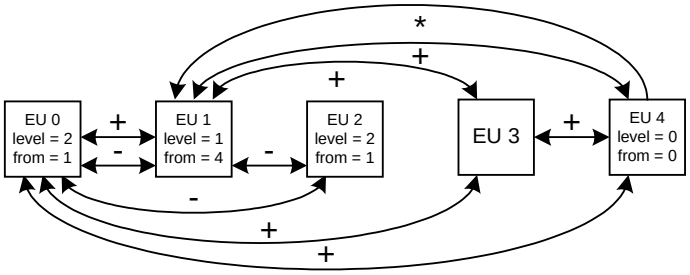


Figure 3-10: Example for the *bfs*-algorithm.

The *bfs* will start at level 0 in node 4. Please note that *from*[4] is undefined at that time. Then, node 1 gets the level 1 and *from*[1] is set to 4. The *bfs* continuous at level 1 in node 1. Then nodes 0 and 2 will get level 2. Moreover, the *from*-attribute of both nodes is set to 1. Finally, at level 2, the node $f = 4$ is reached from node 0. At this time, the *from*-attribute of node 4 is set to 0.

With the reminder *from* the path is reconstructed and a legal permutation w' for instruction w is computed. Let $v \in G$ be the first node of the goal that is reached with the *bfs*. This node is returned by listing 3-1 together with the reminder as pair (v, *from*). Then the path is reconstructed from the sink to the source node f by using the backward links in the reminder *from*. The permutation is computed by moving each operation on that path from execution unit *from*[n] to execution unit n. The simple loop for computing the permutation is shown in listing 3-2.

```
function permutate(w, (v,from), f)
  for each i ∈ EUS do w'(i) ≔ w(i); od
  temp := w(v);
  do
    w'(v) := w(from[v]);
    v := from[v];
  while(f ≠ v)
  w'(v) := temp
```

Listing 3-2: Computation of a legal permutation.

If a legal permutation was computed for each instruction of the user application, then the intermediate super rebinding graph must be updated to become a super rebinding graph reflecting correctly the current fault state of the processor. For this reason, all outgoing edges of node f that are labeled with a faulty operator $o_1,\ldots,o_m$ are removed. For example, in figure 3-9 (b) all outgoing edges of node 4 that are labeled with a multiplication must be deleted. The resulting intermediate super rebinding graph is shown in figure 3-9 (c). After this step the super rebinding graph is valid for the current fault state of the processor, because it reflects correctly the situation that any operation of type $o_1,\ldots,o_m$ cannot be executed on EU *f*. Therefore, another fault that occurs later on can be handled, by using this super rebinding graph during the execution of the self-repair algorithm.

From this property follows immediately that faults that occur simultaneously in different execution units can be handled as consecutive faults in an arbitrary order. Let $f_1,\ldots,f_n$ denote the execution units that become defective in a VARP processor, and let S_0 denote the super rebinding graph for the non-faulty processor. For each faulty EU f_i a corresponding intermediate super rebinding graph S'_i is constructed. Thereby S'_i is obtained from the super rebinding graph S_{i-1} by removing all ingoing edges of the node f_i that are labeled with a faulty operator from f_i. For each $i \in \{1,\ldots,n\}$, the super rebinding graph S_i reflects accurately the fault state of the processor regarding the faults in the execution units $f_1,\ldots,f_i$. The super rebinding graph S_i is obtained from S'_i by removing all outgoing edges of f_i that are labeled with the faulty operators of f_i. In order to compute a legal permutation of an instruction w, for each faulty execution unit f_i the *bfs*- and *permutate*-algorithms must be executed using the corresponding intermediate super rebinding graph S'_i. I.e., after computing the legal permutation for a single instruction, the super rebinding graph S_0 must be restored for computing the legal permutation of the next instruction in the user application. This is avoided by maintaining all intermediate super rebinding graphs during the adaptation of the user application. The complete *software-based rebinding* algorithm is shown in listing 3-3. Thereby w an instruction and *fsEU* the fault state of the processor.

```
function rebind(w, fsEU)

  Let f1,…,fn be the faulty execution units in the processor (determined by fsEU)
  Construct intermediate super rebinding graphs S1', S2',…, Sn' from S0
  for k=1 to n do
    (v,from) := bfs(w,fk,Sk')
    if v = failed then
      exit()
    fi
    w = permutate(w, (v,from), fk)
  od
  return w;
```

Listing 3-3: Software-based Rebinding algorithm.

A major limitation of the presented rebinding algorithm is that it will only work for instruction set architectures in which all operations are single-cycle operations. The reason for that limitation is that the rebinding is computed for each instruction separately. A rebinding of an operation made in a particular instruction does not affect the binding of operations in other instructions. But if there is a multi-cycle operation that needs n clock cycles for being executed, then this operation is executed in n consecutive instructions by the same slot. Thus, changing the binding of this operation in one instruction also affects all other instructions. A solution for overcoming this limitation is presented section 3.3 by using a more complex rescheduling algorithm.

3.2.4 Executing the Repair Routine

Now it is described how the software-based rebinding algorithm from listing 3-3 is executed by the VARP processor, even when the VARP processor is faulty. In order to make sure that the rebinding algorithm is executed properly under most fault situations, there are four versions sr_0, sr_1, sr_2 and sr_3 of the rebinding algorithm. Thereby the version sr_i uses only slot i of the VARP[17] processor. I.e., all non-NOP operations are executed in slot i, while all other slots execute only NOP operations. Thus, the rebinding routine can be executed properly as long as at least a single slot of the VARP processor is faultless. In order to determine the fault state of the VARP processor, suppose a self-test program is used, similar to the one used in the work of Koal [92]. This test program is also available in four different versions tp_0,...,tp_3. Thereby, tp_i executes operations only in slot i for determining the fault state of slot i. When tp_i does not terminate, then some faults during the execution of some control flow operations occurred. In order to cope with this problem without modifying the VARP core, the arbiter is involved in the startup process of the VARP processor.

The arbiter already has access to the program memory of the VLIW core, and it gets control about the reset signal of the VARP processor. In order to invoke the test program tp_i, the arbiter writes a jump instruction for a branch to the first instruction of tp_i into the program memory at address 0x0. Then the arbiter resets the VARP processor that will now jump to the first instruction of tp_i. If tp_i terminates, then the test program sends the fault state produced by the self-test routine to the arbiter, simply by writing it into a particular memory address by using a similar protocol as for initiating the read- and write-mode of the arbiter. Moreover, the arbiter uses a timer for observing the runtime of each self-test program. When the arbiter receives a fault state that indicates a fault in slot i or the runtime is exceeded, then slot i is considered as faulty from the arbiter.

Using this principle, the arbiter invokes all test programs. After that it determines a faultless slot i according to the received fault states. Then, a branch operation that jumps to the first instruction of the rebinding routine sr_i is written into program memory address 0x0, and a reset signal is send to the VARP processor. Thus, the VARP processor executes sr_i that can check the fault state. The fault state is available at a reserved data memory addresses, such that it can be easily accessed with the memory operations of the VARP processor.

[17] Please recall that each slot can execute branch and memory operations.

3.2.5 Fault Tolerant Code Generation

As it can be noticed in listing 3-2 the proposed rebinding algorithm will not always be successful in finding a legal rebinding. For example, consider the instruction in figure 3-11 that should be executed in the given data path. This is the same data path configuration as in figure 3-7 (a). As in the previous example, it is assumed that the multiplier in EU 4 is faulty. Then there exists no legal permutation of the operations. This follows immediately from the fact that the multiplication must be executed by EU 1, but EU 1 must execute a subtraction, because the three subtract operations can be executed only by EU1, EU2, and EU3.

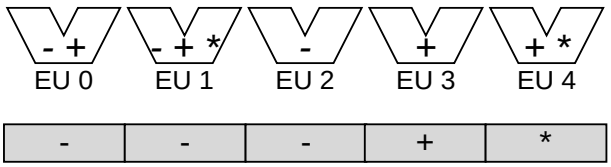


Figure 3-11: Example of a not permutable instruction.

In such a situation the rebinding algorithm will fail to find a legal permutation. In order to avoid the occurrence of this situation in the field, it must be ensured that the rebinding algorithm will always find a legal permutation. For this reason, each instruction must be created in such a way that, for a specified set of fault states of the VARP processor, the rebinding algorithm is able to find a legal permutation. I.e., already the compiler or assembler programmer that generates the binary code of the user application must take care of these specified fault states. In order to specify this set of fault states in a compact way, the classification of fault states into k-slot-faults and k-operator-faults from definition 2-4 and definition 2-5 respectively is used.

3.2.5.1 Slot Faults

Please recall that a k-slot-fault is a fault state where exactly k slots of the VARP processor are faulty.

Definition 3-3: ($\leq m$-slot-fault)
Any k-slot-fault is also an $\leq m$-slot-fault for $m \in SLOTS$, if $k \leq m$.

This means that up to m slots cannot be used anymore for executing operations, when the VARP processor is affected by a $\leq m$-slot-fault. In such a situation the rebinding algorithm will find a legal permutation if and only if there are at least m NOP operations in each instruction, such that these NOP operations can be scheduled to the defective execution units. This property is defined as the m-unit-fault property.

Definition 3-4: (m-unit-fault property)
An instruction w has the *m-unit-fault property*, if it contains at least m NOP operations.

This limitation can be easily incorporated into the scheduling algorithm used by a compiler for a VLIW processor, such that each instruction of the generated binary code has the m-unit-fault property. The compiler simply generates the code for a VLIW architecture with $|SLOTS| - m$ slots and not for a VLIW architecture with $|SLOTS|$ slots. Thus, m slots in each instruction are unused and the rebinding algorithm will always find a legal permutation for each $\leq m$-slot-fault in the field.

For example, in order to cope with any fault state that is a $\leq$2-slot-fault, the binary code for the VARP processor is generated in such a way that slots 2 and 3 execute only NOPs. I.e., they can be considered as spare resources. When a fault affects slot 1 in the field, then all operations executed in slot 1 are moved from the rebinding algorithm into slot 2. As a consequence, the rebinding algorithm can be considered as a purely software-based method for administrating the available hardware redundancy. Therefore this is an active hardware redundancy method where the administration of hardware resources is done in software. However, in order to handle $\leq m$-slot-faults, m slots are preserved in the processor architecture as spare slots that cannot be used for executing operations in the originally generated schedule. Thus, there is a considerable loss of performance in the processor. This loss of performance is quantified in section 3.2.6.

3.2.5.2 Operator Faults

Please recall that a k-operator fault affects exactly k operators in an arbitrary set of execution units. As a consequence the affected execution units cannot execute operations with the type of the affected operator.

Definition 3-5: ($\leq n$-operator-fault)
Any k-operator-fault is also a $\leq n$-operator-fault for $n \in \mathbb{N}$, if $k \leq n$.

For example, if an adder in EU 2 and a multiplier in EU 3 of a data path are faulty, then this fault state is a $\leq$2-operator-fault. It will be also a $\leq$3-operator-fault, but not a $\leq$1-operator-fault.

In order to generate a schedule that can be modified by the rebinding algorithm in such a way that an arbitrary $\leq n$-operator-fault can be handled, the following n-operator-fault property must hold for each instruction w in the schedule.

Definition 3-6: (n-operator-fault property)
An instruction w has the *n-operator-fault* property, if

$$\left|\{t \mid i \in EUS \wedge t = type(w(i)) \wedge t \neq NOP\}\right| + \left|\{i \mid i \in EUS \wedge type(w(i)) = NOP\}\right| \geq n+1\,.$$

I.e., the operations that are scheduled into a single instruction w must have at least n+1 different types, whereby each *NOP* in w is counted as a separate type. For example, an instruction that contains two *NOPs* and two *add*-operations has the 2-operator-fault property, but not the 3-operator-fault property. An instruction that contains two *add*- and two *mul*-operations has the 1-operator-fault property. In general, for a VARP processor with k issue slots, no operation can have a n-operator-fault property with $n \geq k$, because each operation can have at most k different operation types.

Theorem 3-3

Respecting the n-operator-fault property in an arbitrary instruction w guaranties the existence of a legal permutation for the operations in w for each $\leq n$-operator-fault.

Proof:

Let $e_0,\ldots,e_m$ with $m < n$ denote all the execution units with faulty operators, and let f_i denote the number of faulty operators in execution unit e_i. Because there are at most n faulty operators allowed for an $\leq n$-operator fault, for $f_0,\ldots,f_m$ holds:

$$\sum_{z=0}^{m} f_z < n+1\,.$$

It is now proven that to each e_i an operation in w can be assigned that can be executed on e_i. In e_0 there are $f_0 \leq n$ faulty operators. These faulty operators do not include a NOP. Hence, $|\mathcal{O}| - f_0 > 0$ faultless operators exist in e_0. But w contains at least n+1 operations with different types (when counting each NOP as a separate type). Therefore, at least one of these operations can be executed by e_0. If now an arbitrary execution unit e_i, with $0 \leq i \leq m$ is considered, then for e_i holds that

1. at most i operations of different type were assigned to $e_0,\ldots,e_{i-1}$, and
2. f_i operators are faulty in e_i.

Thus, there are at most

$$i + f_i$$

operation types that must be excluded from being assigned to e_i, because i operations with different type were already assigned to $e_0,\ldots,e_{i-1}$ and maybe

another f_i operation types are faulty in e_i. However, because $f_z \geq 1$ for all z with $0 \leq z \leq m$, it follows:

$$i + f_i = \sum_{z=0}^{i-1} 1 + f_i \leq \sum_{z=0}^{i-1} f_z + f_i = \sum_{z=0}^{i} f_z < n + 1.$$

Thus, at least one operation exists in w that can be assigned to e_i, because the operations assigned to $e_0,\ldots,e_{i-1}$ and the operations that cannot be assigned to e_i will have at most n different types.

◊

Both properties, the m-unit-fault property and the n-operator-fault property, can be incorporated easily into a scheduling algorithm for generating code for a statically scheduled processor architecture like the VARP processor. In listing 3-4 it is shown how both properties can be integrated into a list scheduling algorithm, yielding a *fault-tolerant list scheduling algorithm* that can generate a *fault tolerant schedule*. A list scheduling algorithm is often used for scheduling purposes in compilers. It uses a priority list. The operations from the priority list are scheduled in priority order into the current instruction w. If no more operations can be scheduled into w, then the next instruction is generated. Both, m-unit-fault property and n-operator-fault property can be easily checked during scheduling operations into the current instruction by counting the NOP operations in w and the different operation types in w.

```
(N,E) ... directed acyclic graph of the basic block
m     ... tolerable m-slot-faults
n     ... tolerable n-operator-faults
while not all nodes in N are scheduled do
  create next empty instruction w
  Compute ready-list r
  for each node in r in priority order do
    schedule operation r into instruction w if
    - no dependencies are violated and
    - there is a free resource for r and
    - at least m EUs are unused in w after scheduling r into w and
    - w contains at least n+1 different operation types after scheduling r into w
  od
  emit instruction w
od
```

Listing 3-4: Fault tolerant list scheduling algorithm that checks for the m-unit-fault property and for the n-operator-fault property.

Please note that generating a fault tolerant schedule for $m = 0$ and $n > 0$ allows for using all execution units in each instruction. For example, an instruction of a four-issue VARP processor that contains two *add*-, a single *mul*-, and a single *shift*-operation will have the 2-operator-fault property. This seems to be an advantage over the unit-fault property. However, although all slots can be used in a homogeneous data path with $|SLOTS|$ slots, n operators of each type are used for spare purposes. This results from the fact that each instruction must contain at least $n + 1$ different operation types. Thus, for any type t, there are in each

instruction at least n operations that must have a different type from t. Therefore at least n operators of type t are unused in each instruction. Moreover, if another component of the slot than the execution unit is faulty, then this fault cannot be handled anymore. Please recall that such a fault can be handled only by avoiding the usage of the complete slot, and this requires at least a schedule that has the 1-unit-fault property.

3.2.6 Results

3.2.6.1 Hardware Overhead

The proposed software-based rebinding does not cause any hardware overhead in the VARP processor itself. However, when the processor does not provide access to its program memory, then the embedded system that contains the processor must be extended by an arbiter for instructions transfer between data and program memory. As shown in table 3-1, the size of the arbiter described in section 3.2.2 is around 3.5% of the size of the non-fault tolerant VARP processor. For comparison the sizes of an execution unit and a slot in relation to the full VARP processor are provided, too. The arbiter has approximately the size of a simple 16-bit integer unit.

VARP Processor	EU	Slot	Arbiter
100%	4%	16.5%	3.5%

Table 3-1: Size of the arbiter in relation to the size of the non-fault tolerant VARP processor.

However, there are also VLIW-like architectures that have access to their program memory without using an arbiter. For example, Intel's Itanium processor has a unified memory, but separate program and data caches for avoiding structural hazards in the pipeline. For such processor architectures the software-based rebinding can be applied without creating hardware overhead in the system that contains the processor.

3.2.6.2 Performance Degradation

The impact of the fault-tolerant scheduling algorithm (see listing 3-4) on the performance of the processor when executing several benchmark programs is investigated in this sub-section. Thereby the performance degradation depends on the data dependencies and the available instruction level parallelism of the used benchmark. The worst case, when handling $\leq m$-slot-faults, will occur for basic blocks, where the non-fault tolerant schedule has the following form:

- No instruction contains a NOP, i.e., the benchmark provides high instruction level parallelism.

- All operations of an instruction are data dependent on the operations in the preceding instruction. I.e., delaying the scheduling of an operation also delays the scheduling of all operations in successive instructions.

Every operation u that is executed in such a non-fault tolerant schedule before operation v, must be also executed before operation v in the fault tolerant schedule. When $\leq m$-slot-faults in a VARP processor with N slots should be tolerated, then each instruction must contain at most $N - m$ non-NOP operations. Because each instruction in the non-fault tolerant schedule contains N non-NOP operations, it must be replaced by

$$k := \left\lceil \frac{N}{N-m} \right\rceil$$

instructions in the fault tolerant schedule. Thus, in the worst case, the runtime of the fault tolerant schedule is k times the original runtime.

In a similar way the worst case runtime for handling $\leq m$-operator faults is obtained. The non-fault tolerant schedule must have the same form as in the previous case. Moreover, all operations in each instruction must have the same type. Then, a $\leq m$-operator fault can be handled only, when each instruction in the fault tolerant schedule contains at least m NOPs, because then this instruction has the m-unit-property. I.e., each instruction must have the same number of NOPs as an instruction for handling $\leq m$-slot-faults in the previous case. Thus, in the worst case, the runtime of the application is increased by the same factor k. Table 3-2 compares the worst-case runtimes obtained with software-based rebinding with the worst case runtime obtained with hardware-based rebinding for a VARP processor with four slots.

	runtime for ≤m-slot-faults		***runtime for ≤m-operator-faults***	
	hardware-based rebinding	**software-based rebinding**	**hardware-based rebinding**	**software-based rebinding**
$m = 0$	100%	100%	100%	100%
$m = 1$	200%	200%	200%	200%
$m = 2$	300%	200%	300%	200%
$m = 3$	400%	400%	400%	400%

Table 3-2: Comparison of worst-case runtimes from hardware-based rebinding and software-based rebinding.

Only for $m = 2$ the worst case runtime of the hardware-based rebinding is improved by the software-based rebinding. However, in practical applications it is very unlikely that the described structure of data dependencies will be present in basic blocks. For this reason the shown worst case overheads will not be achieved in most situations. The benchmark programs from table 2-4 were also used for generating fault tolerant schedules by taking various $\leq n$-operator-faults and $\leq m$-slot-faults into account. The results are shown in table 3-3. The schedules were

generated for a VARP processor with four slots. For each benchmark the obtained schedule length is shown together with the runtime overhead in percent.

Benchmark	m=0 n=0		m=0 n=1		m=0 n=2		m=1 n=0		m=1 n=2	
ARF	8	0%	10	25%	10	25%	10	25%	11	38%
EWF	14	0%	14	0%	16	14%	16	14%	17	21%
FFT	10	0%	11	10%	13	30%	13	30%	15	50%
DCT-DIF	11	0%	15	36%	19	73%	18	64%	21	91%
SWIM1	8	0%	8	0%	11	38%	10	25%	12	50%
SWIM2	5	0%	5	0%	6	20%	5	0%	6	20%
DCT-LEE	14	0%	14	0%	17	21%	17	21%	18	29%
DCT-DIT	14	0%	15	7%	19	36%	18	28%	21	50%
Average	0%		10%		32%		26%		44%	

Table 3-3: Length of schedules tolerating ≤n-operator-faults and ≤m-slot-faults.

The length of the non-fault-tolerant schedule is shown in the column where $m = 0$ and $n = 0$. The non-fault tolerant schedule is generated without taking the m-unit-fault property and the n-operator-fault property into account. These schedules define the base-line lengths. Therefore they have a runtime overhead of 0%. The remaining columns in table 3-3 show the schedule length for each benchmark program, if it is scheduled by using the fault tolerant list scheduling algorithm from listing 3-4 in such a way that n-operator-faults and m-slot-faults can be handled. Furthermore, the performance degradation due to the increased schedule length is given in percent compared with the length of the non-fault-tolerant schedule. The results show that the performance degradation for schedules that tolerate a single operator fault is 0% in the best case (SWIM1) and for some benchmarks up to 36% (DCT-DIF). 1-slot-faults can be handled by accepting a performance degradation in the range of 0% to 64%. Handling either a 2-operator fault or a 1-slot-fault is possible with a performance degradation in the range between 20% and 91%.

Note that the fault-tolerant schedules guarantee the handling of each ≤n-operator-fault and each ≤m-slot-fault. But there are many other fault states with k faulty operators ($k > n$) that can be handled by these schedules, too. For example, there are usually many operators in the homogeneous data path that are rarely utilized by the operations in each instruction. Even when more than n of these operators will fail, a legal permutation will be found with the rebinding algorithm for many situations.

The rebinding algorithm was implemented in assembler. This was possible with 245 instructions per version without instruction transfer. For simplicity, the rebinding algorithm was implemented as a sequential program, i.e., each instruction contains only one operation that is distinct from a NOP. By a worst case analysis of the code of the rebinding routine, a worst case runtime of less than 500 clock cycles for computing the rebinding of a single instruction was determined. On average the required number of clock cycles is much lower. Table

3-4 shows the runtimes in clock cycles for performing the rebinding of various benchmark programs with this rebinding routine.

Benchmark	SW-Rebinding	
	1-operator-fault	1-slot-fault
ARF	2543	2390
DCT-DIF	3961	3445
FFT	3740	3085
EWF	3726	3890
DCT-DIT	4365	4225
DCT-LEE	5010	4085

Table 3-4: Worst-case runtime of the software-based rebinding of the shown benchmark programs in clock cycles.

Taking into account that the VARP processor can run with 500 MHz, rebinding a program with 2^{16} instructions takes less than 0.07 seconds in the worst case, which is acceptable for a self-repair process that is executed during the startup phase.

3.2.6.3 Reliability Analysis

A VARP system with software-based rebinding that can handle all ≤1-slot-faults is considered for the reliability estimation. Such a VARP processor must have at least three functioning slots. Thus, all four slots can be considered as a 3-of-4-system that is connected in series with the other non-fault tolerant components of the VARP processor. All the fault-tolerant components belonging to the 3-of-4-system are gray shaded in table 3-5. Moreover, the size of all components of the VARP processor, including the arbiter, is shown, too. Please note that the fault tolerant part of a slot now also includes the fetch register. Faults in the fetch register can be handled, because the rebinding is done in the program memory, i.e., before the fetch stage.

Component	Fault Tolerant VARP processor												
		Fault Tolerant Part of a Slot								Control Logic	Arbiter	Register File	
			Pipeline Registers			Read Port	Bypass	EU	Fetch Reg				Reg
				DE	WB								
Instances of components	1	4	1	1	1	2	2	1	1	1	1	1	64
Cell area per component	29825	4748	791	446	146	1153	240	1171	199	539	1014	9280	145
Reliability function	-	$R_{slot}(t) = e^{-5539 \cdot \lambda t}$	-	-	-	-	-	-	-	$R_{ctrl}(t) = e^{-539 \cdot \lambda t}$	$R_{rl}(t) = e^{-1014 \cdot \lambda t}$	$R_{rf}(t) = e^{-9280 \cdot \lambda t}$	$R_{rf}(t) = e^{-145 \cdot \lambda t}$

Table 3-5: Cell area and reliability function for each component of the fault tolerant VARP processor, assuming a constant failure rate λ.

Hardware overhead that must be taken into account for the reliability analysis may also occur in the program memory for two reasons: First, storing the rebinding algorithm in the program memory creates some memory overhead. Second, by the fault tolerant scheduling of the user application, the schedule

becomes longer, and additional memory must be provided for these additional instructions. However, the latter problem will not occur, when the instructions are compressed, as it is done in other VLIW architectures, too [108]. Also for the VARP processor a compression scheme has been developed in [47], such that NOP operations must not be stored explicitly in the program memory. When such a compressed instruction is fetched, then a small decoder in the fetch stage restores the required NOPs. Hence, if the fault tolerant list scheduling algorithm inserts additional NOPs into an instruction, then these NOPs do not increase the memory consumption of the user application. Moreover, also the rebinding routine benefits from this compression, because each instruction of the rebinding routine contains only one operation that is different from a NOP. Therefore, the size of the compressed four versions of the rebinding routine is almost equal to the size of 245 non-compressed instructions. This is a small overhead in memory that is neglected for the reliability analysis. The reliability function of a VARP processor with software-based rebinding including the arbiter is given in equation (3-1).

$$R_{swa}(t) = e^{-29825 \cdot \lambda t} + 4 \cdot \left(e^{-10833 \cdot \lambda t} \cdot e^{-4748 \cdot 3 \cdot \lambda t} \cdot \left(1 - e^{-4748 \cdot \lambda t} \right) \right) \tag{3-1}$$

The reliability function for a VARP system, where it is assumed that the VARP core has already access to its program memory, is given by the reliability function in equation (3-2).

$$R_{sw}(t) = e^{-28811 \cdot \lambda t} + 4 \cdot \left(e^{-9819 \cdot \lambda t} \cdot e^{-4748 \cdot 3 \cdot \lambda t} \cdot \left(1 - e^{-4748 \cdot \lambda t} \right) \right) \tag{3-2}$$

Equation (3-2) differs from equation (3-1) by neglecting the size of the arbiter in the serial part of the system. Both functions are plotted in figure 3-12.

In order to determine a reference system for comparison, two scenarios will be considered. First, the software-based rebinding is used for making a non-fault tolerant processor off the shelf fault tolerant. It is assumed that this is the non-fault tolerant VARP processor from figure 2-1. Then the fault tolerance is achieved at the cost of some performance degradation, because the application executed on the non-fault tolerant VARP processor must be scheduled with the fault tolerant scheduling algorithm in such a way that each instruction has the 1-unit-fault property. The reliability function $R_{NFT_4}(t)$ for the non-fault tolerant VARP processor was already given in equation (2-1). For the second scenario, it is assumed that the fault tolerance is obtained by explicitly introducing spare resources. Then the non-fault tolerant VARP processor has three slots only. The fault tolerant VARP processor is obtained by adding a fourth slot as spare resource that is administrated with the software-based rebinding. The reliability function for a VARP processor with three slots is given by

$$R_{3slots}(t) = e^{-24069 \cdot \lambda t}.$$

The reliability functions of all four systems are plotted in figure 3-12 together with the reliability function $R_{hwr}(t)$ from equation (2-4) for the VARP processor with hardware-based rebinding.

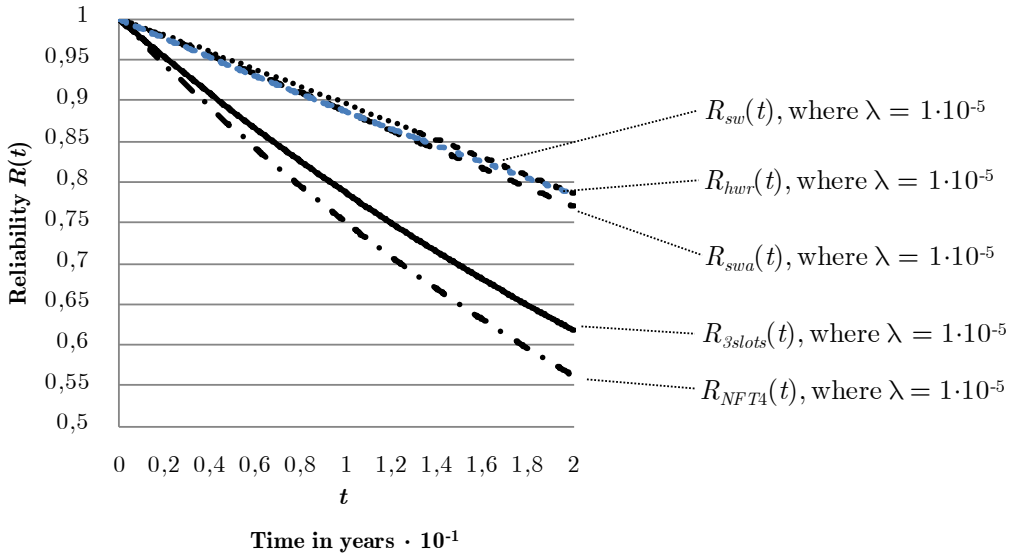


Figure 3-12: Reliability plots for the software-based rebinding.

In any case, the VARP processor with software-based rebinding has a higher reliability than both non-fault tolerant processors (either with four or three slots). The reliability improvement factor (RIF) ranges between 2.6 and 1.7 for the plotted time interval, whereby the highest improvement is achieved for smaller times. It can be noticed that the reliability of the fault tolerant system with hardware-based rebinding (R_{hwr}) does not differ very much from the reliability of the VARP systems with software-based rebinding (R_{sw} and R_{swa}). In particular, when t is very small, then the reliability of both systems with software-based rebinding is a little bit higher than the reliability of the system with hardware-based rebinding (RIF ≈ 1.1). But the reliability of the system with software-based rebinding and arbiter (R_{swa}) becomes a little bit worst than the reliability of the system with hardware-based rebinding for $t > 1$, which represents in the given time scale 10 years. However, the differences are not significantly, such that the reliability of all three systems can be considered as almost equal.

3.2.7 Conclusions

The presented software-based rebinding is a feasible method for implementing active hardware redundancy in statically scheduled superscalar processors, even then when they were not designed to be fault tolerant. The presented software-based rebinding is essentially a simple way for managing redundant computational resources in the superscalar data path of such a processor completely in software

and without introducing any hardware overhead for administration purposes in the core. The administration is accomplished in the field, but in off-line mode. Therefore, this approach can handle permanent faults only. With the presented method almost the same reliability as with a hardware-based administration scheme is achieved. However, the performance degradation is much lower (24% for handling 1-slot-faults when executing the used benchmarks) than the performance degradation caused by the hardware-based rebinding (99% for handling 1-slot-faults when executing the used benchmarks). An interesting property of the software-based approach is that the performance can be traded against reliability. The reliability can be scaled by software – simply by specifying the tolerable operator- and slot-faults. With the selected parameters the application is compiled using the fault tolerant list-scheduling algorithm. This allows for in-the-field handling of the specified amount of faults. The presented approach is well suited for real-time applications, because no performance degradation of the application is introduced by the rebinding routine. This follows from the fact that only the operations within each instruction are permutated by the self-repair routine. The control flow of the program and the length of each basic block remain unchanged. However, the performance penalty is introduced during the fault tolerant scheduling, as it was demonstrated for some benchmarks in section 3.2.6.2. Moreover, this performance penalty must be paid, even then when there is no fault inside of the processor. This can be considered as a disadvantage. For example, the hardware-based rebinding presented in section 2.3 creates performance degradation only when a fault is present in the data path. Another disadvantage of the software-based rebinding (but also of the hardware-based rebinding) is that it is limited to processor architectures with single-cycle operation. Multi-cycle operations cannot be handled correctly with the rebinding routine. On the other hand, this allows for a very simple adaptation scheme that works locally in each instruction. For this reason, it is not necessary to consider any kind of dependencies between operations in different instructions – neither data dependencies nor control flow dependencies. In the next section a more sophisticated software-based administration is presented that also takes dependencies into account. Thereby, operations can be also moved into other instructions and multi-cycle operations can be handled.

3.3 Fine-Grained Self-Repair

This section introduces a software-based self-repair approach that can be used for fault handling in statically scheduled superscalar data paths. It relies on a *rescheduling algorithm* that uses the basic idea presented in figure 3-2, i.e., the user application is modified in the field such that the operations of the application are scheduled around faulty components in the data path. The rescheduling algorithm

is executed in an off-line mode, such that only permanent faults can be handled. The difference from the previously presented software-based rebinding is that operations can be moved between instructions; i.e., a new schedule can be computed for the user application. In order to maintain the control flow dependencies in the program, the rescheduling algorithm works at basic block level. I.e., each basic block of the user application is moved from the program memory into the data memory. There the schedule of the basic block is modified by the rescheduling algorithm according to the current fault state of the processor. The modified basic block, which is now adapted to the current fault state, is written back into the program memory. Due the rescheduling, the length of a basic block may be increased as it is shown in figure 3-2. This requires some more sophisticated adaptations of the user application that are discussed in section 3.3.3. On the other hand, the rescheduling algorithm allows for overcoming some disadvantages of the rebinding algorithm. In particular, it can handle multi-cycle operations, and the granularity of the components, whose reconfiguration can be administrated, is lowered below slot- and execution unit level. First, a coarse grained version of the rescheduling is presented in section 3.3.1. Then this coarse-grained version is extended in section 3.3.2 in order to lower the granularity of the components that can be administrated.

3.3.1 Rescheduling at Slot- and Execution Unit Level

It is assumed that the fault state of the processor is given at slot- and execution unit level by the fault state functions *fsSlot* and *fsEU*. Furthermore, the rescheduling algorithm is introduced first only for single basic blocks that have been already moved from the program memory into the data memory. Then it does not matter, if the length of a basic block is increased. The basic block transfer can be organized simply by using the arbiter presented in section 3.2.2. Then, the ordering of the operations in each instruction corresponds to the ordering presented in figure 3-5. Moreover, if all instructions of a basic block are stored in ascending order in the data memory, then all operations of the basic block are ordered sequentially in the data memory of the VARP processor. Formally this sequential ordering is obtained by considering the instruction sequence w_0, w_1, w_2,..., w_k that forms a basic block of length $k+1$. This ordering implies a total ordering of the operations of that basic block as follows: Let $w_m(i)$ and $w_n(j)$ be two operations, where $i, j \in SLOTS$ and $0 \leq m, n \leq k$. Then $w_m(i) < w_n(j)$, if and only if $m < n$ or $m = n$ and $i < j$. This sequential ordering can be considered as a *sequential schedule* of the basic block with the same data dependencies as in the original schedule.

Theorem 3-4

Given a basic block b by an instruction sequence s, where each instruction in s has the no-input-is-output property from definition 2-3. Then the sequential schedule of b has exactly the same data dependencies as the instruction sequence s.

Proof:

Consider two operations $w_m(i)$ and $w_n(j)$ of an instruction in s. It is obvious that the theorem holds for $m \neq n$, because then the execution order of both operations is not changed by the sequential schedule. For $m = n$, both operations were executed in parallel in s. Thus, no data dependency exists. Without loss of generality suppose $i < j$. Then there is neither a true- nor an anti-dependency between $w_m(i)$ and $w_n(j)$ in the sequential schedule, because the destination register of each operation differs from the source register(s) of the other operation due to the no-input-is-output property. Furthermore, there is no output dependency, because both operations must have different destination registers.

◊

Now, the rescheduling algorithm shown in listing 3-5 uses the sequential schedule as input and produces an instruction sequence respecting all data dependencies from the sequential schedule and the current fault state of the processor.

```
1  instructionCount := 0;
2  while not all operations in the sequential schedule are scheduled do
3    Create a new empty instruction currInstr
4    for each slot ∈ SLOTS do
5      if counter[slot] = 0 then  // Is EU slot still busy with a multi-cycle operation?
6        for reg := 0 to 63 do    // No, then try to find an operation for that EU
7          Def[reg] := 0; Use[reg] := 0;
8        od
9        for each operation v from the sequential schedule in given order do
10         if v is already scheduled in currInstr then
11           Def[v.Dst] := 1; Use[v.Src1] := 1; Use[v.Src2] := 1;
12         fi
13         if v is not scheduled then
14           if CanBeExecuted(v,slot) &&            // Check for fault state
15              Def[v.Src1]=0 && Def[v.Src2]=0 &&   // Check for flow-dependecies
16              Use[v.Dst]=0  &&                    // Check for anti-dependencies
17              Def[v.Dst]=0                        // Check for output-dependencies
18           then
19             counter[slot]  := latency(type(v))
20             currInstr[slot]:= v
21             lastDef[v.dst] := instructionCount;
22             srcSlot[v.dst] := slot;
23             set state of v to scheduled;
24             break;
25           else
26             Use[v.Src1] := 1      // Update lookup-tables, when v cannot be scheduled
27             Use[v.Src2] := 1
28             Def[v.Dst] := 1
29         fi
30       fi
31     od
32   fi
33 od
34 for each slot ∈ SLOTS do
35   if counter[slot] > 0 then counter[slot]--; fi
36 od
37 instructionCount := instructionCount + 1;
38 od
```

Listing 3-5: Software-Based Rescheduling algorithm.

Thereby, the rebinding algorithm tries to parallelize the operations from the sequential schedule. The produced instruction sequence is the modified schedule for the basic block that must be written back into the program memory. Basically the rescheduling algorithm is a list scheduling algorithm that has some similarities with the scoreboard algorithm described in [80]. The scoreboard algorithm is used in dynamically scheduled superscalar processor architectures for dispatching the sequential operation stream to the available functional units. Thereby a scoreboard is used to check efficiently for data dependencies between operations. The rescheduling algorithm in listing 3-5 uses a similar lookup-table-based technique for detecting data dependencies. These lookup-tables are named $use : \text{REGS} \rightarrow \{0,1\}$ and $def : \text{REGS} \rightarrow \{0,1\}$. The modified schedule for a basic block is now generated instruction by instruction with the rescheduling algorithm. For this reason in line 3 of listing 3-5 a new empty instruction is created, which is appended to the modified schedule generated so far. The *for*-loop in line 4 tries to find for each slot of the current instruction an unscheduled operation from the sequential schedule that can be scheduled at this position. For this reason, the operations in the sequential schedule are processed in their given order (line 9) and for each unscheduled operation v it is checked in lines 14 to 17 whether it can be scheduled or not in the current *slot* of *currInstr*. v can be scheduled at this position, if the fault state of the slot allows for the execution of v (this is checked by the function *CanBeExecuted*) and no dependencies are violated (this is checked with the expressions in lines 15 to 17[18]).

If an operation v from the sequential schedule cannot be scheduled at the current position, then in lines 26 to 28 the lookup-tables *use* and *def* are updated. These lookup-tables are used for checking for dependency violations. Suppose there is an operation u in the sequential schedule that writes to register r and reads from registers *s1* and *s2*, and u was not scheduled in *currInstr* at position *slot*. Then $def(r) = 1$, $use(s1)$, and $use(s2) = 1$. Now suppose that some iterations later of the loop in line 9, it must be checked for an operation v, with $u < v$, whether v can be scheduled at position *slot* of *currInstr* or not. Whether or not v violates some dependencies can be checked by using the lookup-tables *use* and *def* in lines 15 and 16. Thereby, for the lookup-table *def* holds in this situation: $def(r) = 1$, if and only if there is an unscheduled operation u in the sequential schedule, with $u < v$, that has operand r as destination operand. Thus, if v has r as source operand, then v cannot be scheduled, because there is a true-dependency from u to v. In line 15 it is checked for this true dependency for both source operands of v. In a similar way anti- and output dependencies are checked. In line 16 it is checked if there is a

[18] For better understanding it is assumed that all operations have one destination register and two source registers.

preceding unscheduled operation u of v ($u < v$) that reads a register r that is written by operation v. If this is the case, then there is an anti-dependency from u to v, and $use(r)$ was set to 1 when u was processed. This is easily checked, when v is processed and scheduling of v is avoided. Output dependencies are checked in line 17. If there is a preceding unscheduled operation u of v that writes to the same register r that is also written by v, then there is an output dependency from u to v. In this situation $def(r)$ was set to 1 in line 28, when u was processed, and the output dependency is detected by the expression in line 17, such that v is not scheduled.

Please note that each operation that is already scheduled into the current instruction *currInst* is treated in lines 10 to 12 as if it were unscheduled. Hence, it is ensured that that no two operations with the same destination register are scheduled into the same instruction. Moreover, the no-input-is-output property is maintained for the new generated instruction.

Data dependencies also arise from memory operations. Determining whether or not there is a data dependency between two memory operations is in general not decidable [10]. Even making this decision for some simple situations may require a complex data flow analysis. For this reason a conservative approach is used that considers two memory operations always as data dependent, if at least one of them is a write operation into the memory. This can be easily incorporated into the rescheduling algorithm by considering the memory as a single register that is written with store-operations and read with load-operations. Then the dependencies between memory operations can be checked in the same way as the dependencies arising from register accesses. In general it can be stated that the implementation of the rescheduling algorithm is simplified, when the instruction set of the processor is designed in such a way that all data dependencies between operations are explicitly visible in the coding of the operations. For example, a bad solution is an instruction set, where each ALU-operation implicitly modifies the status flags of the processor (e.g. carry, zero, etc.). When these implicitly modified flags are used by successive operations, then the resulting data dependencies must be found by a complex data flow analysis of the binary code. Moreover, data dependencies that are made explicitly visible by the instruction encoding, for example by register addresses, can be identified easier from the rescheduling algorithm, when the instruction set encodes these register addresses always in the same bit fields. Otherwise too many different cases must be distinguished by the rescheduling algorithm for extracting these dependencies from the binary code. Hence, a very regular instruction coding as it is common for reduced instruction set computers (RISC) is very beneficial.

Multi-cycle operations are handled by the proposed rescheduling algorithm in lines 5, 19, and 34 to 36. When an operation is scheduled in a particular slot s, then the

latency of this operation is stored in the array *counter*(s). Single cycle operations have a latency of 1. In lines 34 to 36 the counter of each slot is decremented before a new instruction is created. In line 5 a slot is excluded from scheduling, if its corresponding counter is not 0.

In order to use the proposed rescheduling algorithm for fault handling at various granularity levels, only the function *CanBeExecuted* in line 14 must be adapted, such that it takes care of the current fault state of the processor. In the subsequent sections this function is defined for various granularity levels.

3.3.1.1 Fault Handling at Slot Level

In order to handle faults at slot level the expression *CanBeExecuted*(v,s) in line 14 of listing 3-5 is defined as

$$CanBeExecuted(v,s) := fsSlot(s),$$

whereby *fsSlot* is the fault state of the slots of the VARP processor for fault handling at slot level. Please recall that $fsSlot(s) = 0$, if there is a fault in any of the components belonging to slot s. Thus, operation v is only scheduled into slot s, if s is faultless. In figure 3-13 an example is given. The basic block of the original schedule is shown in figure 3-13 (a). Now suppose that the *add*/*sub*-component in the execution unit of slot 1 is faulty; i.e., $fsSlot(0) = fsSlot(2) = fsSlot(3) = 1$ and $fsSlot(1) = 0$. Then the rescheduling algorithm will produce the schedule in figure 3-13 (b). There, only NOPs are scheduled into slot 1. It can be further noticed that the length of the adapted schedule is increased.

Add r0,r2 -> r4	Add r3,r1 -> r5	shl r6 -> r6	mul r0,r1 -> r8
Sub r4,r5 -> r4	div r0,r1 -> r7	shl r6 -> r6	Inc r20 -> r20
Mul r4,r5 -> r0	Sub r4,r1 -> r9	Dec r6 -> r6	Shr r8 -> r8
Add r0,r9 -> r2	NOP	NOP	Inc r20 -> r20

(a)

Add r0,r2 -> r4	NOP	Add r3,r1 -> r5	shl r6 -> r6
mul r0,r1 -> r8	NOP	Sub r4,r5 -> r4	div r0,r1 -> r7
shl r6 -> r6	NOP	Inc r20 -> r20	Mul r4,r5 -> r0
Sub r4,r1 -> r9	NOP	Dec r6 -> r6	Shr r8 -> r8
Add r0,r9 -> r2	NOP	Inc r20 -> r20	NOP

(b)

Figure 3-13 (a) Example of an original schedule. (b) Schedule that is obtained from the schedule in (a) after adaptation at slot level.

3.3.1.2 Fault Handling at Execution Unit Level

In order perform the rescheduling algorithm for fault handling at execution unit level, the function *CanBeExecuted* is defined as

$$CanBeExecuted(v,s) := fsEU(s,v) \wedge fsSlot(s),$$

whereby *fsEU* is the fault state of the execution units and *fsSlot* is the fault state of the slots for the execution unit level. Please recall that at execution unit level a slot is considered as operational, if all faults are located in the execution unit of

that slot and at least a single operator of the execution unit is operational. Thus, an operation v can be scheduled into slot s, if there is no faulty component in that slot or all faulty components are operators in the execution unit of that slot which are not required for executing the operation v. Two examples are shown in figure 3-14.

Add r0,r2 -> r4	NOP	Add r3,r1 -> r5	shl r6 -> r6
Sub r4,r5 -> r4	mul r0,r1 -> r8	div r0,r1 -> r7	shl r6 -> r6
Inc r20 -> r20	Mul r4,r5 -> r0	Sub r4,r1 -> r9	Dec r6 -> r6
Shr r8 -> r8	NOP	Add r0,r9 -> r2	Inc r20 -> r20

(a)

Add r0,r2 -> r4	NOP	Add r3,r1 -> r5	shl r6 -> r6
mul r0,r1 -> r8	NOP	Sub r4,r5 -> r4	div r0,r1 -> r7
shl r6 -> r6	NOP	Inc r20 -> r20	Mul r4,r5 -> r0
Sub r4,r1 -> r9	NOP	Dec r6 -> r6	Shr r8 -> r8
Add r0,r9 -> r2	NOP	Inc r20 -> r20	NOP

(b)

Figure 3-14: Schedule that is obtained from the schedule in figure 3-13 (a) after adaptation at slot level.

The schedule in figure 3-14 (a) is obtained from the original schedule in figure 3-13 (a) for a fault state in which only execution unit 1 cannot execute *add*/*sub*-operations, but the *mul*-operator in slot 1 can be used. The resources are much better utilized in this example than in the schedule in figure 3-13 (b). However, if, for example, the fault in slot 1 is located in a read port of slot 1, then the whole slot 1 cannot be used anymore, and the schedule in figure 3-14 (b) is obtained after rescheduling at execution unit level.

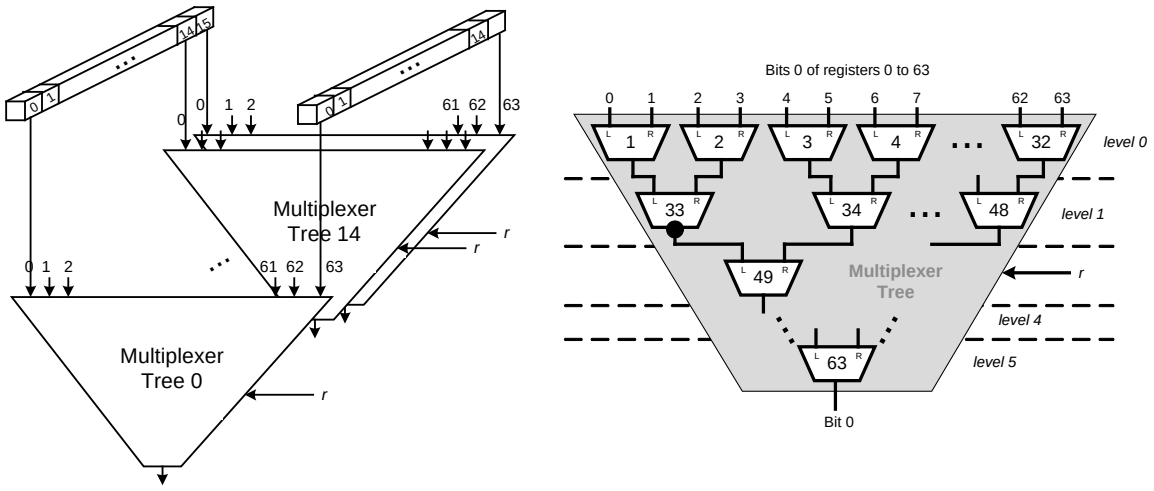
3.3.2 Lowering the Granularity

Performing the rescheduling at execution unit level forces the self-repair algorithm to avoid the usage of the whole slot, if there is a defect in any other component than the execution unit. Because an execution unit of the VARP processor only makes up 25% of the size of a slot (see for example table 3-5), it is more likely that a fault occurs in another component of the slot. For this reason a finer-grained fault handling for the remaining components of a slot is considered now.

3.3.2.1 Fault Handling at Read Port Level

The VARP processor has eight read-ports in order to support simultaneous access of all slots to the register file. Each read port of the register file has nearly the same size as an execution unit. Thus, the probability that a defect is located in a read port is almost equal to the probability of a fault in the execution unit. Therefore, a finer-grained self-repair approach should explicitly account for defects in the read ports, such that a slot with a faulty read-port can still be used for executing operations. This is achieved by avoiding the usage of the defect part of a read port.

Each slot of the VARP processor contains two read ports. A single read-port of a register-file with 16-bit registers consists of 16 multiplexer trees; one tree for each output bit of the read-port, as it is shown in figure 3-15 (a). The multiplexer tree k selects bit k of that register whose value should be read through this port. Thus all multiplexer trees of the same read port receive the same control signal r. The control signal of the left/right read port in slot s is the value *src1*/*src2* from the fetch-register of slot s (see figure 2-1). Each multiplexer tree is built up from 2:1 multiplexers as it is shown in figure 3-15 (b). Multiplexers 1 to 32 belong to the lowest level of the tree and multiplexer 63 to the highest level.



(a) (b)

Figure 3-15: (a) Structure of a read-port in the register file. (b) Structure of a single multiplexer tree from (a).

According to [69] the faults that affect a multiplexer structure like the read port can be classified into *data faults* and *address faults*. When an address fault is present, then the wrong input is selected by the multiplexer. Such a fault may occur due to a stuck-at fault in the control signal r. A data fault affects the data inputs and/or output of a multiplexer. For example, in figure 3-15 (b) a stuck-at fault at the data output of multiplexer 33 is shown. This fault prevents the correct reading of bit 0 from the registers 0 to 3. But bit 0 of all other registers can be accessed correctly. In a similar way, an address fault in a read port prevents the correct reading of a particular subset of registers, but in most situations at least some registers can be accessed correctly. It is obvious that there are faults, effecting that the whole read-port cannot be used anymore. For example, a data fault at the output of multiplexer 63. However, it is much more likely that a fault occurs in one of the multiplexer that belongs to a lower level, because there are much more multiplexers in the lower levels. Thus there is a good chance to have functioning access to some registers via a read-port, even if there are some faults in this read port.

Please recall that the fault state of a read port is given by the fault state function *fsRP*, and that $fsRP(p,r) = 1$, if and only if register r can be accessed correctly through read port p. I.e., the fault state is given at a functional level, no matter whether the fault is a data- or address fault. With this fault state function also multiple faults in a single read port are covered. For example, the fault state function for the data fault of read port p shown in figure 3-15 (b) is defined as

$$fsRP(p,r) = \begin{cases} 0, & \text{if } r < 4 \\ 1, & \text{otherwise} \end{cases}$$

Please note that the example in figure 3-15 (b) only shows a single multiplexer tree of a read port. It is assumed that all other multiplexer trees of the same read port are faultless in this example.

For fault handling, the usage of a faulty sub-components in a read port p is avoided by accessing only registers r through the read port p, for which $fsRP(p,r) = 1$. I.e., the rescheduling algorithm in listing 3-5 must schedule only such operations into slot s that use registers that can be accessed correctly through the used read port. This is incorporated in the rescheduling algorithm by defining the function $CanBeExecuted(v,s)$ from listing 3-5 as

$$\begin{aligned} CanBeExecuted(v,s) := {} & fsEU(s,v) \wedge fsSlot(s) \wedge \\ & (fsRP(2 \cdot s,\ src1(v)) \vee \neg usesLRP(type(v))) \wedge \\ & (fsRP(2 \cdot s + 1,\ src2(v) \vee \neg usesRRP(type(v))). \end{aligned}$$

In this definition $src1(v)$ is the register number of the left operand, $src2(v)$ is the register number of the right operand. The functions $useLRP : \mathcal{O} \rightarrow \{0,1\}$ and $useRRP : \mathcal{O} \rightarrow \{0,1\}$ determine for a given operation type, whether this operation type uses the left respectively the right read port or not. It is $usesLRP(t) = 1$, if and only if operations of type t uses the right read port. Now suppose that operation v uses the left, but not the right read port. Then the expression $\neg usesRRP(type(v))$ becomes true and the fault state of the right read port, given by *fsRP*, becomes superfluous for the evaluation of the expression. I.e., whether operation v can be scheduled to slot s depends in this situation only on the fault state of the left read port and on the fault states of the execution unit and the slot.

An example for the application of the rescheduling algorithm at read port level on the schedule in figure 3-14 (a) is given in figure 3-16.

Add r0,r2 -> r4	NOP	shl r6 -> r6	Add r3,r1 -> r5
Sub r4,r5 -> r4	mul r0,r1 -> r8	shl r6 -> r6	div r0,r1 -> r7
Inc r20 -> r20	Mul r4,r5 -> r0	Sub r4,r1 -> r9	Dec r6 -> r6
Shr r8 -> r8	NOP	Inc r20 -> r20	Add r0,r9 -> r2

Figure 3-16: Schedule from figure 3-14 (a) adapted at read-port level.

It is assumed that registers *r0* and *r1* cannot be accessed via the left read-port of slot 2. Moreover, the *add*/*sub*-operator in slot 1 is still faulty. I.e., the left operand of all operations, which are scheduled to slot 2, must be distinct from *r0* and *r1*. This property is violated by the *div*-operation and one *add*-operation in the schedule in figure 3-14 (a). The rescheduling at read port level exchanges the operations from slot 2 and 3 in the second and fourth instructions. The obtained schedule in figure 3-16 executes the program correctly without reading *r0* and *r1* through the left read port of slot 2 and without using the *add*/*sub*-operator from slot 1. Note that the handling of both faults at slot level would result in a schedule that contains only NOPs in slots 2 and 3. Thus the schedule in figure 3-16 utilizes the available resources much better. However, if there is a fault in a bypass of a slot, then this fault must be handled at read port level by declaring the whole slot as faulty. This limitation is eliminated by self-repair at bypass-level described in the subsequent section.

3.3.2.2 Fault Handling at Bypass Level

In this section it is shown how a slot can be used for executing operations, even if there are faults in its bypasses. Moreover, it will be shown how this slot can be used, even when its read port(s) are completely failing; i.e., no register can be accessed through the read port(s).

Each slot has a left and a right bypass. The left bypass provides the left operand and the right bypass the right operand (see figure 2-1) for the execution unit. A bypass selects either

- the value from its corresponding read port, or
- one of the four values from the execution stage (these are the values *ex0*,...,*ex3* at the outputs of each execution units EU_0,...,EU_3),
- or one of the four values from the write-back stage (these are the value *wb0*,...,*wb3* at the outputs of the four write-back registers).

Thus a bypass has the same structure as a read port for a register file with nine registers. The structure of a single multiplexer tree of a bypass is shown in figure 3-17 together with the multiplexer tree from the corresponding read port.

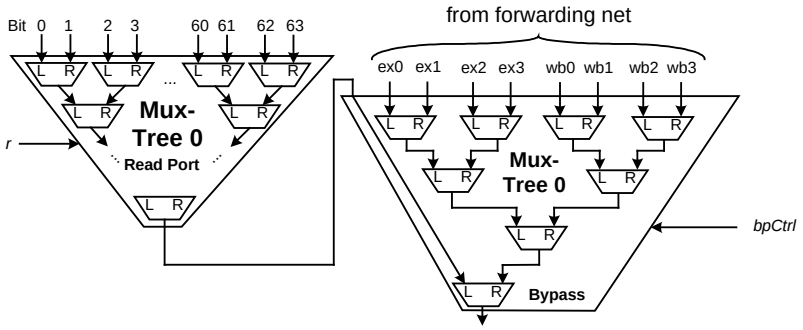


Figure 3-17: Structure of a single multiplexer tree of a bypass and its connection with the multiplexer tree of the corresponding read port.

The input that must be selected from the bypass is binary coded into the *bpCtrl* signal. This signal is generated from the bypass control logic. The bypass control must decide whether forwarding is needed or not. When no forwarding is needed, then the multiplexer in the highest level of the bypass multiplexer tree must select the value from the read port, otherwise it must select that value from the forwarding network. Each multiplexer tree of the bypass can have the same data and address faults as a read port. Moreover, a fault in the bypass control logic will also cause an address fault. The functional misbehaviour produced by either of these faults is modelled by the bypass fault state functions *fsBPex* and *fsBPwb* that were defined in section 2.2.2. Please recall that the meaning of these functions for s, $d \in SLOTS$ is:

- If $fsBPex(s, 2 \cdot d) = 0$, then the value from the execution stage of slot s cannot be read correctly through the left read port of slot d.
- If $fsBPex(s, 2 \cdot d + 1) = 0$, then the value from the execution stage of slot s cannot be read correctly through the right read port of slot d.
- If $fsBPwb(s, 2 \cdot d) = 0$, then the value from the write-back register of slot s cannot be read correctly through the left read port of slot d.
- If $fsBPwb(s, 2 \cdot d + 1) = 0$, then the value from the write-back register of slot s cannot be read correctly through the right read port of slot d.

The functional misbehaviour that in slot d a value from the read port is not read correctly through the bypass, when forwarding is not needed, is modelled by the read port fault state function *fsRP*. Then for all $r \in$ REGS $fsRP(2 \cdot d,\ r) = 0$ if the left bypass is faulty, and $fsRP(2 \cdot d + 1,\ r) = 0$, if the right bypass is faulty. A fault in the bypass is now handled by avoiding the usage of the faulty bypass. This is achieved in two ways:

1. Either an operation is moved into another slot with a working bypass, or
2. the distance between the producer of an operand and the consumer of an operand is changed.

In order to determine the distance between an operation that produces a value and an operation that consumes a value, it is assumed that the instructions of a basic block are enumerated. Then the number of the last instruction that defines register r must be remembered for each register r. This is accomplished by the function *lastDef* that can be defined easily in the rescheduling algorithm in listing 3-5 (see line 21). There, for each scheduled operation v with destination register r, *lastDef*(r) is set to the number of the current instruction (*instructionCount*). Moreover, for each register r the slot in which this register has been lastly defined is remembered by the function $srcSlot : \text{REG} \rightarrow \text{SLOT}$, whose value is set in line 22 in listing 3-5. If an operation v, which reads the register r, is scheduled some instructions later, then the distance between the producer u of the value in r and the consumer v of the value in r is determined by *instructionCount* – *lastDef*(r). This distance is used to determine whether the value of r is found in the register file, in the execution stage or in the write-back stage. Therefore, the following three cases are distinguished:

1. *instructionCount* – *lastDef*(r) = 1: Then u is in the execution stage, while v is in the decode stage. Thus, the value of r is located in the execution stage and the bypass provides the register value of r from the execution stage.
2. *instructionCount* – *lastDef*(r) = 2: Then u is in the write-back stage, while v is in the decode stage. Thus, the value of r is located in the write-back stage and the bypass provides the register value of r from the write-back stage.
3. *instructionCount* – *lastDef*(r) > 2: Then the processing of u in the pipeline is finished, and the value of r is already stored in the register file. The bypass provides the register value of r from the read port.

According to the fault state functions *fsBPex* and *fsBPwb*, an operation v cannot be scheduled into slot d of the current instruction

- if its left operand is produced from an operation u that was scheduled with a distance of 1 into slot s and *fsBPex*(s, $2 \cdot d$) = 0 or
- the right operand is produced from an operation u that was scheduled with a distance of 1 into slot s and *fsBPex*(s, $2 \cdot d + 1$) = 0 or
- its left is produced from an operation u that was scheduled with a distance of 2 into slot s and *fsBPwb*(s, $2 \cdot d$) = 0 or
- the right operand is produced from an operation u that was scheduled with a distance of 2 into slot s and *fsBPwb*(s, $2 \cdot d + 1$) = 0.

These conditions are incorporated into the function *CanBeScheduled* in listing 3-5 as follows by the lines 4 and 5 for the left operand and lines 9 and 10 for the right operand of an operation v:

```
1  CanBeExecuted(v, s) :=
2        fsEU(s,v) ∧ fsSlot(s) ∧
3        ( ¬usesLRP(type(v)) ∨
4          instructionCount -lastDef[src1(v)] = 1 ∧ fsBPex(srcSlot(src1(v)), 2·s) ∨
5          instructionCount -lastDef[src1(v)] = 2 ∧ fsBPwb(srcSlot(src1(v)), 2·s) ∨
6          instructionCount -lastDef[src1(v)] > 2 ∧ fsRP(2·s, src1(v))) ∧
7
8        ( ¬usesRRP(type(v)) ∨
9          instructionCount -lastDef[src2(v)]=1 ∧ fsBPex(scrSlot(src2(v), 2·s+1) ∨
10         instructionCount -lastDef[src2(v)]=2 ∧ fsBPwb(srcSlot(src2(v), 2·s+1) ∨
11         instructionCount -lastDef[src2(v)] > 2 ∧ fsRP(2·s+1,src2(v)))
```

Line 2 ensures that the operation can be executed by the execution unit of slot s. Lines 3 to 6 will check whether or not the left operand can be provided correctly for the operation, and lines 8 to 11 will check this for the right operand. The expressions in lines 3 and 8 make sure that the expressions from line 3 to 6 respectively 8 to 10 become true, if the corresponding operand (left respectively right one) is not used from the current operation. In that case the operation can be scheduled into the current slot, even if the bypass and/or read port are faulty. The expressions in line 6 (for the left operand) and line 11 (for the right operand) become active, if the operand is read from the register file, i.e. the distance between producer and consumer is larger than 2. In that case it is checked whether or not the requested register can be accessed correctly through the corresponding read port.

For example, suppose operation v uses register *r1* as left operand, register *r7* as right operand, and should be scheduled into slot 2. Furthermore, *r1* has been defined from an operation that was scheduled in the previous instruction in slot 3 (i.e., *instructionCount* - *lastDef*(1) = 1 and *srcSlot*(*r1*) = 3) and *r7* was defined three instructions before (i.e., *instructionCount* - *lastDef*(7) = 3). Thus, the value of *r1* is found in the execution stage of slot 3 and the value of *r7* is found in the register file. This means that the left bypass in slot 2 must provide the input value *ex3* correctly and the right bypass must provide the value from the corresponding read port (see figure 3-17). Now the expression *CanBeScheduled*(v, *2*) becomes true, if and only if *fsBPex*(3, 2 · 2) in line 4 and *fsRP*(2 · 2 + 1, 7) in line 11 are both true. If this is not the case, then v is not scheduled in slot 2.

Please note that an operation that reads a register may be scheduled into a slot s, even if the register cannot be accessed correctly through the used read port in that slot. This is always possible, if the needed value is available in a pipeline register, and the forwarding from that pipeline register to slot s is functioning. Thus, the bypass can be used to feed the execution units directly with data from the same or other slots via the bypass without using a faulty read-port.

In figure 3-18 an example is given for a schedule that is obtained from the schedule in figure 3-16 by taking into account that the complete read-port for the left operand in slot 0 is faulty (i.e., $fsRP(0, r) = 0$ for all $r \in REGS$). Thus, only operations whose left operand can be obtained from a pipeline register can be scheduled to slot 0. For this reason, in instruction 1 a NOP was scheduled to slot 0. The value of registers *r4* in instruction 2 is obtained from the execution stage, because it was computed in instruction 1. Similarly, the value of *r6* is computed in instruction 2 and is read in instruction 3 from the execution stage, too. The value of *r8* is computed in instruction 2 and is read in instruction 4 from the write-back stage. Thus, all left operands for operations in slot 0 can be loaded from the pipeline registers, and no access to the register file via the faulty read-port is necessary.

NOP	*shl r6 -> r6*	*Add r3,r1 -> r5*	*Add r0,r2 -> r4*
Sub r4,r5 -> r4	*mul r0,r1 -> r8*	*shl r6 -> r6*	*div r0,r1 -> r7*
Dec r6 -> r6	*Mul r4,r5 -> r0*	*Sub r4,r1 -> r9*	*Inc r20 -> r20*
Shr r8 -> r8	*NOP*	*Inc r20 -> r20*	*Add r0,r9 -> r2*

Figure 3-18: Schedule adapted at bypass level.

The proposed rescheduling algorithm works fine for each basic block separately. But the register allocation for the user application is usually done globally in order to avoid unnecessary load- and store-operations in each basic block. Therefore, a value may be written into register r in a particular basic block, and the same value is read in another basic block. This may creates problems for the rescheduling of the first two instructions of each basic block, because a basic block b may have more than one control flow predecessors. Suppose that each of the control flow predecessors writes to the same register, which is accessed in the first two instructions of basic block b. Such a situation is shown in the control flow graph in figure 3-19.

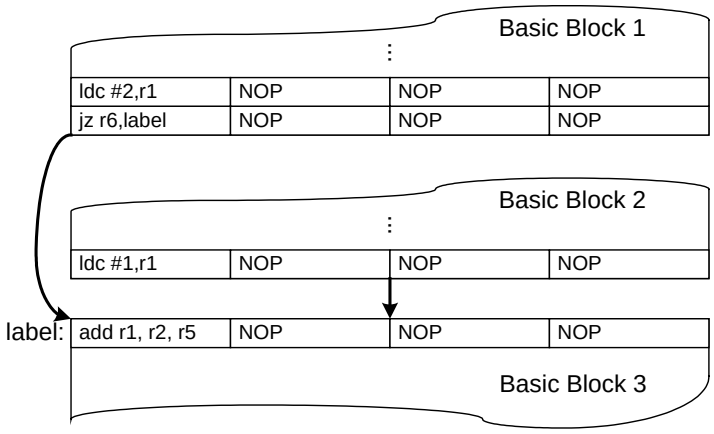


Figure 3-19: Example, where the location of the value in *r1* in basic block 3 depends on the actual control flow.

There register *r1* is defined in both control flow predecessors of basic block 3, and register *r1* is accessed in the first instruction of basic block 3. Depending on the control flow predecessor, from which basic block 3 is entered, the value of *r1* is either found in the register file (when control flow reaches from the basic block 1) or in the execute-stage (when control flow reaches from the basic block 2). Thus, for both situations it must be checked that register *r1* can be accessed correctly, when the *add*-operation is scheduled by the rescheduling algorithm at the shown position. This requires knowledge about all control flow predecessors of each basic block, which would dramatically increase the complexity of the rescheduling algorithm. For this reason another strategy is used, where only the *direct control flow predecessor* a of each basic block b must be known. This is the basic block that can enter b without taking a branch operation. In figure 3-19 basic block 2 is the direct control flow predecessor of basic block 3. For all control flow predecessors that enter basic block b via a taken branch, it is ensured by resetting the pipeline that all values written in these control flow predecessors into a register are found in the register file, when the first instruction of b is executed. Now suppose that basic block b is entered from a direct control flow predecessor a. Then the first instruction of basic block b is located at address x, and the last instruction of basic block a must be located at address $x-1$. Please note that the predecessor block a may contain a conditional branch in its last instruction, but this branch is not taken. The sequential ordering of basic blocks a and b implies that the basic blocks are processed in the given sequential ordering by the rescheduling algorithm. Thus basic block a has been processed by the rescheduling algorithm immediately before basic block b is processed. Therefore, the registers defined in the last two instructions of basic block a can be remembered from the rescheduling algorithm. During the rescheduling of basic block b, none of these registers is allowed to be a source operand in the first two instructions of basic block b. With this restriction all registers accessed in b within the first two instructions are located in the register file. The restriction can be incorporated efficiently into the rescheduling algorithm by introducing an additional function $minInstr : REGS \rightarrow \mathbb{N}$. This function specifies for each register the first instruction in which this register is allowed to be accessed. For a register r that is defined in the last instruction of basic block a is $minInstr(r) := 2$, and for a register r defined in the second last instruction of a, it is $minInstr(r) := 1$. The complete function *minInstr* for a basic block b is easily obtained from the *lastDef*-function of the previously scheduled basic block a by

$$minInstr_b(r) := \begin{cases} 0, \text{ if } lastDef_a(r) \leq instructionCount_a - 2 \\ lastDef_a(r) - instructionCount_a + 1, \text{ otherwise} \end{cases}.$$

Thereby, $lastDef_a$ is the function as it is defined in listing 3-5 after executing the rescheduling algorithm for the preceding basic block a. $instructionCount_a$ is the number of instructions in basic block a after rescheduling (see listing 3-5).

Now an operation with source operand r can be scheduled into instruction *currInst* of basic block b only, if $instructionCount \geq minInstr(r)$. This restriction is incorporated in the function *CanBeExecuted* in lines 4 and 10 as follows:

```
1   CanBeExecuted(v, s) :=
2       fsEU(s,v) ∧ fsSlot(s) ∧
3       ( ¬usesLRP(type(v)) ∨
4         instructionCount ≥ minInstr(src1(v)) ∧
5         (instructionCount -lastDef[src1(v)] = 1 ∧ fsBPex(srcSlot(src1(v)), 2·s) ∨
6         instructionCount -lastDef[src1(v)]=2 ∧ fsBPwb(srcSlot(src1(v)), 2·s) ∨
7         instructionCount -lastDef[src1(v)] > 2 ∧ fsRP(2·s, src1(v)))) ∧
8
9       ( ¬usesRRP(type(v)) ∨
10        instructionCount ≥ minInstr(src2(v)) ∧
11        (instructionCount-lastDef[src2(v)]=1 ∧ fsBPex(scrSlot(src2(v), 2·s+1) ∨
12        instructionCount -lastDef[src2(v)]=2 ∧ fsBPwb(srcSlot(src2(v), 2·s+1) ∨
13        instructionCount -lastDef[src2(v)] > 2 ∧ fsRP(2·s+1,src2(v))))
```

The given definition of the function *CanBeScheduled* can be implemented efficiently in assembly for the VARP processor by using simple arrays. Thus, the complexity was not increased very much compared with the complexity of that function in the coarse grained implementation.

3.3.2.3 Register-Level

All of the software-based techniques presented so far will not help to cope with faults in the register file. For example, it is not possible to replace a faulty register by another register in the program code. However, the 64 registers of the VARP processor including the write-ports makes up about 30% of the processor. Thus, it is very likely that a fault will occur in the register file. The structure of a single register k, which also includes the logic for writing a value into this register, is shown in figure 3-20.

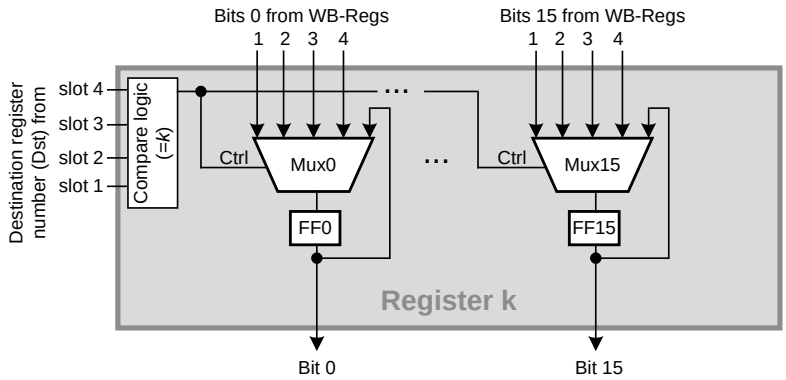


Figure 3-20: Implementation of a single register of the register file including the write-port.

Bit m of the register is stored in flip-flop m (FF m). The input for flip-flop m is selected by the multiplexer *Mux* m. This is either the output of flip-flop m (meaning that the register keeps the current value) or bit m of one of the four write-back registers. The compare logic ($=k$) determines the control signal of these multiplexers. For this reason, the compare logic compares the destination register number from the write-back stage of each slot (bit-fields *dstWb* in the write-back registers in figure 2-1) with k. If one of these register numbers matches with k, then the corresponding control signal for the multiplexers is generated.

A fault that appears in such a register may affect the compare logic, the multiplexer, or the flip-flops of the register itself. In each case a wrong value may be loaded into the corresponding register. This may happen, too, even if the register should not be loaded. This will wrongly overwrite the value in the register. In any case, the fault state of the register file is modelled by the register fault state function *fsRF*, where $fsRF(r) = 0$ means that register r is faulty. The usage of such a faulty register is avoided by never using the data from such a register. In order to avoid the usage of a register, a global register renaming can be done by the rescheduling algorithm from listing 3-5. I.e., when the original schedule of the user application is generated, for example by a compiler, then a few registers in the register file are preserved as backup registers. These registers are not used in the original user application. When a fault occurs in one of the used registers, then each usage of such a register is replaced by the usage of a backup register. This replacement can be done in the scheduling algorithm from listing 3-5 in line 20. There the used registers in operation v are checked, and, if necessary, replaced by a backup register before the operation is scheduled into the new instruction. Because the register renaming is done globally, this functionality is easily implemented by using a simple mapping $rename : REGS \rightarrow REGS$ that maps each functioning register r on register r, and each faulty register r' on a backup register distinct from r'. In section 3.3.4 it is shown by simulations that a small number of backup registers is sufficient for most situations.

3.3.3 Relocation of Basic Blocks

This section describes how the prolongation of the schedule of a rescheduled basic block is treated. Because the rescheduling algorithm can move operations into other instructions, the adapted schedule of a basic block may become longer than the original one. This will create the following problems:

1. The adapted schedule of the basic block may not fit into the place of the original schedule. Thus, the instructions that follow the original schedule must be relocated in the program memory.

2. The relocation of instructions will also change the target addresses of some branch operations.
3. In order to cope with the first two problems, the basic block boundaries must be known.

Solutions for these problems were presented by Müller and Schölzel in [124]. In the subsequent sections it is briefly described, how basic block boundaries can be reconstructed, how the relocation of instructions is done, and how the patching of branch target addresses will work.

3.3.3.1 Reconstruction of Basic Block Boundaries

A simple solution for determining the basic block boundaries in the binary code of the user application is a lookup-function t that stores for each basic block i of the application its length $l(i)$. Knowing the start address s of the first basic block and making the assumption that the basic blocks are consecutively numbered by their sequential order in the program memory beginning with 0, the start address of basic block i is obtained by

$$s + \sum_{k=0}^{i-1} l(k)\,.$$

This solution is simple, but consumes maybe too much memory depending on the application, because l must be kept in memory. For example, having an application with 10.000 basic blocks requires at least 10.000 bytes of data memory[19] that are permanently occupied by the function l.

A more sophisticated method is the reconstruction of the basic block boundaries from the binary code of the user application. According to definition 2-2 a new basic block starts after each branch-operation and at each target address of a branch operation. In order to determine statically the target addresses from the program code, they must be given as constants in the branch operations. Register indirect branches are not allowed, except for returning from a function call. Moreover, it is assumed that all target addresses are absolute addresses. Program counter relative branches can be handled, but they are not discussed here, because the VARP processor does not support relative branches.

The reconstruction of basic block boundaries is integrated into the relocation of instructions. In order to have enough buffer memory for the prolonged schedules of the basic blocks, the complete user application is moved first at the end of the program memory before the rescheduling for all basic blocks starts. During this

[19] Assuming that no basic block is longer than 255 instructions.

process the basic block boundaries are determined. The concept for the used memory layout is shown in figure 3-21.

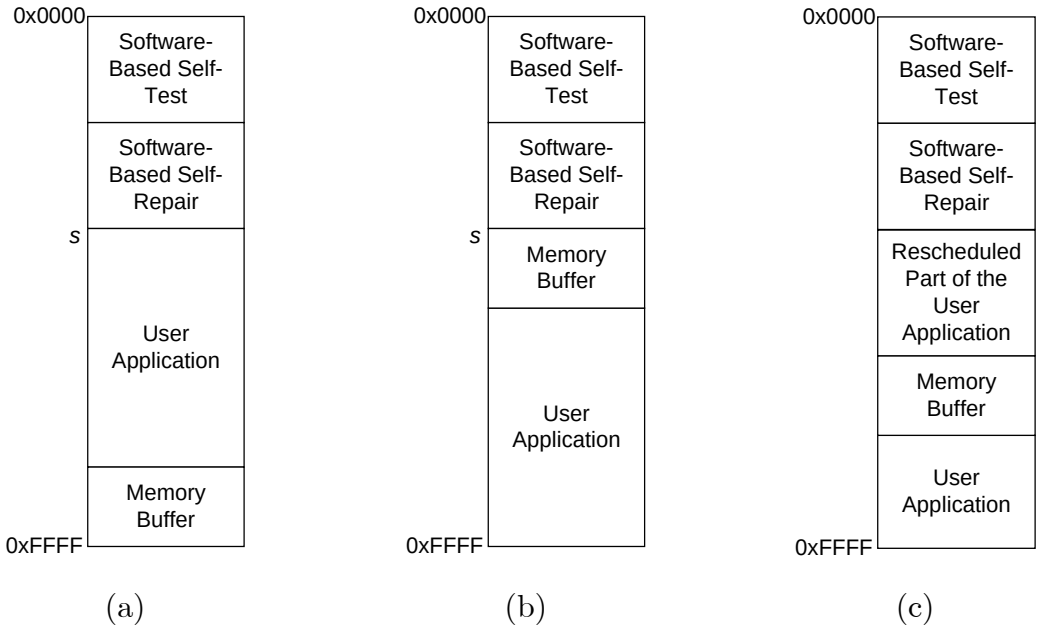


Figure 3-21: (a) Memory layout at startup time. (b) User Application was moved to the end of the program memory. (c) A prefix of the user application has been adapted to the current fault state and moved back to the original location.

The normal memory layout is shown in figure 3-21 (a). In order to have the buffer memory at the beginning of the memory area of the user application, the user application is moved to the end of the program memory as shown in figure 3-21 (b). This is done by reading small fragments of the user application into the data memory using the arbiter protocol from section 3.2.2. Then each fragment is written back into the program memory at address $z + b$, where b is the size of the buffer memory and z is the original address of the fragment. Before each fragment is written back into the program memory, it is scanned for branch operations. Each branch operation at address x that branches to the target address y creates the beginning of basic blocks at addresses $x + 1$ and y. These addresses will be collected in an ascending order in the array s. After collecting all start addresses in array s, $s(i)$ denotes the start address of basic block i. This will create dynamically a similar memory consumption as the simple approach using a static lookup function l, but the array s must not be stored permanently.

3.3.3.2 Relocation of Basic Blocks

The relocation of the basic blocks is strongly coupled with the execution of the rescheduling algorithm in listing 3-5. A single fragment of the backward shifted user application is moved from the program memory into the data memory. Thereby the already gathered basic block information in s is used such that only complete basic blocks are moved from program to data memory. After that a

sequence of basic blocks is located in the data memory. Using the basic block information in s, the rescheduling algorithm is called for each basic block in that fragment. The rescheduling algorithm adapts the current schedule of the basic block to the current fault state of the core. Thereby the rescheduled basic block is written into a new location of the data memory. After rescheduling a single basic block, the prolongation e of that basic block is known. The accumulation of all prolongations obtained so far results in the relocation of the next basic block. This displacement is remembered by a function $bbDispl : \mathbb{N} \rightarrow \mathbb{N}$, where $bbDispl(i)$ gives the displacement of basic block i. After processing all basic blocks of the current fragment, this fragment is written back into the program memory at the first free address of the memory buffer. After that the probably shortened memory buffer is located behind the lastly written fragment, and the next unprocessed fragment is moved from the program into the data memory. This situation is shown in figure 3-21 (c).

Please note that there are various reasons, why the self-repair routine may fail in this stage:

1. The rescheduling algorithm may fail, because it is not able to find a legal schedule that contains all operations of the original basic block. This can happen, when the current fault state of the processor prevents the scheduling of particular operations. This limitation is discussed in more detail in section 3.4.

2. The prolongation of the rescheduled basic blocks becomes longer than the length of the memory buffer. Thus the rescheduled fragment will not fit into program memory without overwriting another basic block.

In both situations the self-repair method fails and the system has a failure; i.e., the occurred fault(s) cannot be covered anymore with the used fault tolerance method.

3.3.3.3 Patching Branch Instructions

When all basic blocks are relocated, then the patching of branch operations can start. For this reason the last instruction of each basic block is moved into the data memory again. Only these instructions will contain branch operations. When such an instruction contains a branch-operation with target address d, then d must be patched. In order to do so, the lookup-table s is used for determining the basic block i for which holds:

$$s(i) = d$$

Please note that the lookup-table s contains the original start addresses of all basic blocks, and d is the branch target according to these original addresses. The displacement of basic block i is given by $bbDispl(i)$. Thus, the target address of the

branch operation must be changed to $d + bbDispl(i)$. Please note that i can be determined efficiently with a binary search, because the entries of the lookup-table s are given in ascending order. After patching the branch operation, the instruction is written back into the program memory.

3.3.4 Results

In this section the performance degradation and reliability improvement of the software-based rescheduling are presented. In particular, the benefits of the fine-grained self-repair capabilities provided by the rescheduling are compared with the capabilities of the previously presented coarse-grained methods.

3.3.4.1 Performance Degradation

The software-based rescheduling algorithm has in common with the hardware-based rebinding algorithm that both approaches cause a graceful performance degradation, when there are faults present in the processor. The software-based rebinding, on the other hand, can handle a specified set of fault states without causing a performance degradation. However, when software-based rebinding is used, then the performance degradation is introduced by the fault tolerant scheduling. Table 3-6 compares the worst-case performance degradations of the software-based rescheduling (SC) with the worst-case performance degradations of the hardware-based rebinding (HR) and with the performance degradations caused by the software-based rebinding (SR) for particular k-slot- and k-operator-faults. I.e., faults are handled either at slot level or execution unit level. According to the results shown in table 3-6, the runtime overhead of the software-based methods is much lower than the overhead caused by the hardware-based method. However, an interesting observation is that the runtime overhead of the software-based rebinding is sometimes a little bit lower than the runtime overhead of the software-based rescheduling, although the rescheduling has more degrees of freedom for moving operations. This result comes from the fact that the used rescheduling algorithm relies on a very simple heuristic that, for example, does not take into account the mobility of an operation, as it can be done by the fault tolerant scheduling. By this, some bad decisions are made by the rescheduling algorithm. These bad decisions increase the length of the adapted schedule. However, improving the simple heuristic will also improve the complexity of the rescheduling algorithm, which may increase significantly its runtime. A further examination of this trade-off was not done within the frame of this work, but could be part of future research, maybe leading to a more complex compiler backend that can be embedded into a system-on-chip for re-compiling the user application. Moreover, the presented runtime overhead for the software-based rescheduling in table 3-6, is the worst case for a particular k-operator-fault/k-slot-fault. In most

cases there was only a single fault state among all fault states belonging to the set of a particular k-operator-fault/k-slot-fault for which this worst case was achieved. For all other fault states the corresponding runtime overhead was lower. In several cases it was even lower than the runtime overhead caused by the software-based rebinding.

Fault Class	**1-operator-fault**			**1-slot-fault**			**2-operator-fault**			**2-slot-fault**		
Method	**HR**	**SC**	**SR**	**HR**	**SC**	**SR**	**HR**	**SC**	**SR**	**HR**	**SC**	**SR**
ARF	63%	25%	25%	100%	25%	25%	125%	25%	25%	200%	75%	88%
DCT-DIF	82%	13%	36%	100%	38%	64%	145%	64%	73%	200%	118%	91%
FFT	70%	30%	10%	100%	30%	30%	140%	90%	30%	200%	130%	80%
EWF	93%	7%	0%	93%	7%	14%	150%	36%	14%	171%	36%	36%
DCT-DIT	64%	21%	7%	100%	21%	28%	114%	36%	36%	200%	71%	79%
DCT-LEE	79%	21%	0%	100%	28%	21%	157%	43%	21%	193%	86%	79%
Average:	**75%**	**20%**	**13%**	**99%**	**25%**	**30%**	**139%**	**49%**	**33%**	**194%**	**86%**	**76%**

Table 3-6: Comparison of the worst-case runtime overhead caused by hardware-based rebinding (HR), software-based rebinding (SR) and software-based rescheduling (SC).

The comparison in table 3-6 does not take into account the benefit of the software-based rescheduling, when faults are handled at read-port and bypass level. For example, a slot fault in table 3-6 means that the faulty slot cannot be used, no matter which self-repair method is used, and no matter which component of the slot is faulty. However, when the faulty component is a read port or a bypass, then the slot containing the faulty components can be used for some operations, if the fine-grained rescheduling is used for adapting the user application. The effect of this granularity reduction is shown by the following example. Four faults were injected in the VARP core:

- The first fault affects the adder of slot 1.
- The second fault affects the left bypass of slot 0 (forwarding from the EX-stage of any slot to the DE-stage is faulty).
- Two faults affect the left read port of slot 3 (register 0 to 3 and 4 to 5 cannot be accessed through this port).

Thus, only slot 2 is faultless. Table 3-7 shows the runtime of three benchmark programs after adapting them to this fault state at different granularity levels with the software-based rescheduling algorithm. I.e., the performance degradation is expressed in terms of required runtime relative to the runtime of the benchmarks on a faultless VARP processor. Thereby the runtime on the faultless VARP processor is 100%.

Benchmark	Runtime after applying software-based rescheduling at the following granularity level		
	Slot	**Execution Unit**	**Bypass and Readport**
ARF	350%	200%	125%
FFT	380%	270%	110%
EWF	243%	193%	121%
DIT	343%	200%	109%
DIF	373%	190%	114%
LEE	350%	193%	114%
Average	**340%**	**208%**	**116%**

Table 3-7: Runtime of the rescheduled benchmark programs after adapting them using different granularity levels.

When the repair routine handles faults only at slot level, then only slot 2 can be used for the execution of the benchmark programs. This results on average in a runtime of 340%. If faults are handled at slot- and execution unit level, then slot 2 can be used and slot 1 can be used for executing multiplications. Slots 0 and 3 cannot be used. This causes on average a runtime of about 208%. Slot 0 and 3 can be used, too, if faults are handled at bypass level and read port-level (runtime on average 116%). This shows that by a fine-grained self-repair the graceful performance degradation can be reduced substantially, because faulty slots can be used for carrying out operations that could not be used if only a coarse-grained self-repair scheme is applied[20].

The proposed rescheduling algorithm was implemented in assembler for the VARP processor. The runtime grows quadratically with the basic block length. Table 3-8 shows the worst case runtime of the rescheduling algorithm in clock cycles, for adapting the shown benchmark programs. For comparison table 3-8 also shows the runtimes of the software-based rebinding routine for these benchmarks from table 3-4.

Benchmark (Length)	1-operator-fault		1-slot-fault	
	rescheduling	**rebinding**	**rescheduling**	**rebinding**
ARF (8)	44655	2543	45296	2390
DCT-DIF (11)	65554	3961	87338	3445
FFT (10)	66771	3740	77116	3085
EWF (14)	125901	3726	132782	3890
DIT (14)	116064	4365	117124	4225
LEE (14)	109168	5010	123304	4085

Table 3-8: Worst-case runtime of the software-based rescheduling in clock cycles for adapting particular benchmarks.

[20] The author has published in [135, 166] also techniques based on scalable algorithms that can be used to trade runtime of the algorithm against service quality. With these techniques the service quality can be reduced such that after an adaptation of the schedule, which has increased the schedule length, the real time constraints may be met.

It can be noticed that the runtime of the rescheduling is more than a magnitude of order larger than the runtime of the rebinding algorithm. This has a strong impact on the runtime needed for adapting a larger user application. This is illustrated by the following example: The sum of the lengths of the benchmark programs in table 3-8 is 71 instructions (see table 2-4). The execution of the rescheduling algorithm for these benchmarks takes 1.17 milliseconds, when a 1-slot-fault is present, and a clock rate of 500 MHz for the VARP processor is assumed. Assuming that an application with 2^{16} instructions is composed of basic blocks similar to the benchmarks in table 3-8, the total runtime will be $(2^{16} / 71) \cdot 1.17$ milliseconds = 1.08 seconds. This is approximately 28 times the adaptation time needed by of the software-based rebinding for the same user application. This example shows that increasing the complexity of the used heuristic in the rescheduling algorithm, in order to improve the quality of the generated schedule as discussed above, must be done carefully. Otherwise the runtime for adapting the user application will exceed very soon an acceptable amount of time.

3.3.4.2 Reliability Improvement

In contrast to the software-based rebinding, the software-based rescheduling does not require to specify the tolerable faults in advance. Assuming that the startup phase is organized by the VARP processor itself, as it was described for the software-based rebinding in section 3.2.4, and that no register renaming is performed, the VARP processor with rescheduling is operational as long as the register file and the control logic are faultless, and there is at least one faultless slot. The same property was assumed for calculating the reliability of the hardware-based rebinding in section 2.4.2. Hence, the reliability function $R_{sca}(t)$ for the VARP processor with software-based rescheduling, but without register renaming, is obtained in the same way as the reliability function in equation (2-4). The adopted reliability function for $R_{sca}(t)$ is given by

$$\begin{aligned} R_{sca}(t) &= R_{ctrl}(t) \cdot R_{rf}(t) \cdot R_{arbiter}(t) \cdot (1 - (1 - R_{slot}(t))^4) \\ &= e^{-10833 \cdot \lambda t} \cdot (1 - (1 - e^{-4748 \cdot \lambda t})^4) \end{aligned} . \qquad (3\text{-}3)$$

This formula reflects the situation that all four slots form a parallel system, and this parallel system forms a serial system together with the control logic, the register file, and the arbiter. $R_{sca}(t)$ is plotted in figure 3-22 for a failure rate of $\lambda = 1 \cdot 10^{-5}$. The reliability $R_{swa}(t)$ of the VARP processor with software-based rebinding is also plotted in figure 3-22, and it is quite similar to the reliability of the VARP processor with software-based rescheduling. Please recall that R_{swa} is the reliability of a VARP processor that can handle faults only in a single slot. Hence, as in the hardware-based rebinding approach, software-based fault handling in multiple slots will not result in a significantly higher reliability than fault

handling in a single slot only, except the system is exposed to a significantly higher failure rate, as it will be shown in figure 3-23.

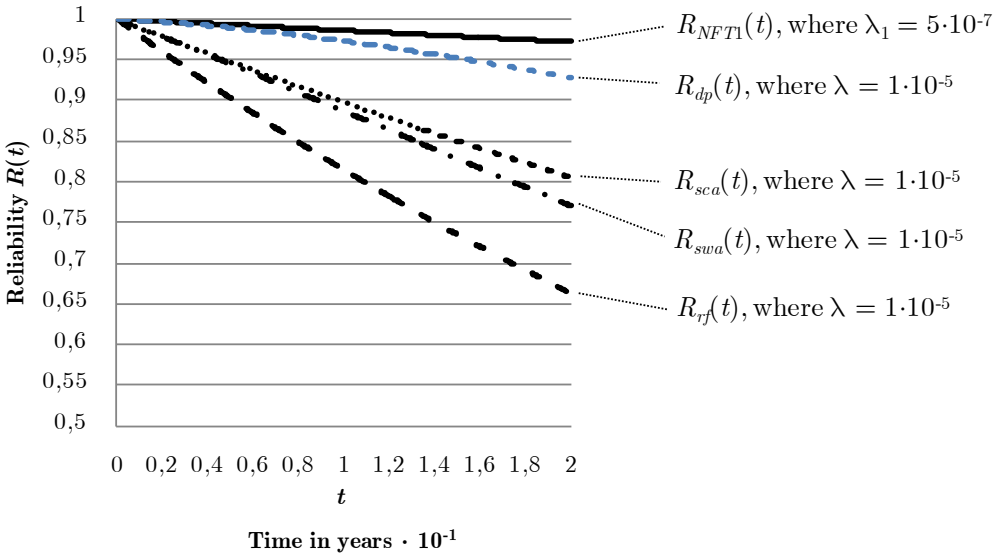


Figure 3-22: Reliability plots for the software-based rescheduling with register renaming (R_{dp}) and without register renaming (R_{sca}).

Figure 3-22 shows also the reliability plot for a VARP processor with software-based rescheduling and a fault tolerant register file. The register file becomes fault tolerant, if register renaming is performed by the rescheduling algorithm as described in section 3.3.2.3. Thereby, each register forms a single component, whose reliability function is given in table 3-5. If five registers of the register file are preserved as backup registers, then the register file of the VARP processor becomes a 59-of-64 system, whose reliability is computed according to equation (1-20). The reliability function $R_{dp}(t)$ for the VARP processor with software-based rescheduling and with register renaming is plotted in figure 3-22 for a failure rate of $\lambda = 10^{-5}$. Thereby, the reliability reaches almost the reliability R_{NFT1} of a non-fault tolerant VARP processor, whose reliability function is plotted for $\lambda = 5 \cdot 10^{-7}$ (see also figure 2-6). I.e., by employing the software-based rescheduling with register renaming a 20 times higher failure rate is acceptable for the fault tolerant system, while maintaining the reliability.

The plotted reliability function $R_{rf}(t)$ in figure 3-22 is the reliability function for the VARP processor, when only register renaming is performed. I.e., only faults affecting the registers of the register file can be handled, and faults affecting any component of the slots cannot be handled. It can be noticed that R_{rf} is clearly below the reliability of a VARP processor, where slot-faults can be handled, e.g., R_{sca}. Hence, fault handling in slots is more beneficial for the VARP processor than fault handling only in the register file. However, fault handling only in the register file increases the reliability compared with a non-fault tolerant version of the

processor and it can be implemented in a much easier manner than fault handling in slots, because only a global register renaming in the user application must be done. Moreover, if the statically scheduled processor has a larger register file, for example with 256 registers, while the data path remains small, then fault handling only in the register file, may become a viable simple solution for improving the reliability. At least software-based register renaming can be also employed in processors that have a simple scalar data path only. In such processors the registers of the register file will be the only inherently available redundant components.

Now it is shown that fault handling in up to three slots becomes superior to fault handling in a single slot for higher failure rates. Consider a failure rate of $\lambda = 1 \cdot 10^{-3}$ as it was also used for the plots in figure 2-15. As already discussed in section 2.4.2, handling of multiple faults in systems that are affected by such a high failure rate is only beneficial, if the portion of the non-fault tolerant components in the system is very low. This is achieved for the VARP processor by making the register file fault tolerant. By this only the arbiter and the control logic are non-fault tolerant components in the VARP processor. Both together constitute only 5.2% of the cell area. The reliability plot for handling faults in up to three slots (R_{dp3}) is compared with the reliability plot for handling faults only in single slot (R_{dp1}) in figure 3-23.

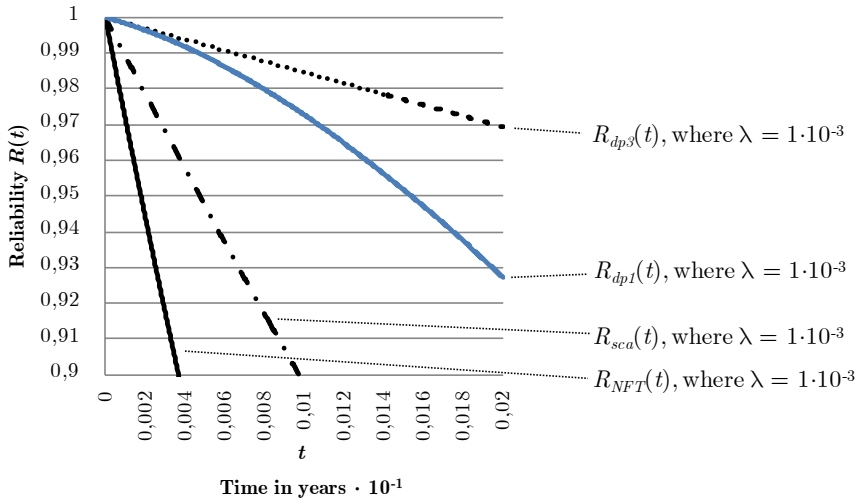


Figure 3-23: Reliability plot for the VARP processor with fault handling at register level and high failure rate.

In figure 3-23 can be also noticed that fault handling for such high failure rates is only beneficial, if the ratio of non-fault tolerant components is small. R_{sca} is the reliability plot of the VARP processor, if faults are handled in up to three slots, but not in the register file. Then the ratio of non-fault tolerant components in the VARP processor rises to 36%, and the achieved reliability is only slightly better than the reliability of a non-fault tolerant VARP processor (R_{NFT}).

Now the benefit of the fine-grained rescheduling is considered, when the VARP processor is affected by multiple faults as it can happen for larger failure rates. For this reason it is allowed that all slots are affected by some faults. But for each operation at least one slot must exist, where this operation can be executed. As a consequence the startup process must be organized by an external entity, because the assumption of a single faultless slot that can be used for the startup process is no longer valid. For example, when the VARP processor is embedded into a multi-core system-on-chip, then another faultless processor in the system adapts the user application for the VARP processor. Such a system was presented in [124]. Another system configuration with a small service core, is presented later in chapter 5.

The probability that the VARP core is running after the occurrence of multiple faults is increased significantly by using the fine-grained rescheduling instead of the coarse-grained rescheduling. This is shown by a fault injection experiment. In this experiment, 10 faults were randomly injected in 1.000.000 VARP processor models, assuming an equal distribution of these faults over the cell area of the core. I.e., larger components have a higher probability of being affected by a fault. After that it was determined whether the VARP processor is still able to execute a user application, when the user application is adapted by one of the software-based rescheduling strategies. When rescheduling takes place at coarse-grained slot level without register renaming, then a VARP processor is operational, if all 10 faults are located in the components of the slots, and there is at least one faultless slot left. I.e., the register file, the control logic, and one slot must be faultless. The upper part of table 3-9 shows that for this scenario only 0.35% of the processors remain operational.

Rescheduling is done at slot level without register renaming							
Operational VLIWs	**VLIWs with faulty Ctrl**	**VLIWs with faults in the register file**	**Percentage of the VLIWs where all faults are in k slots**				
			$k = 0$	$k = 1$	$k = 2$	$k = 3$	$k = 4$
0.35%	17.2%	81.45%	0%	0%	0.01%	0.34%	1.0%
Rescheduling is done at slot level with register renaming							
Operating VLIWs	**VLIWs with faulty Ctrl**		**Percentage of the VLIWs where all faults are in the register file and in k slots**				
			$k = 0$	$k = 1$	$k = 2$	$k = 3$	$k = 4$
44.58%	17.2%		0.001%	0.28%	7.36%	36.94%	33.93%

Table 3-9: Surviving VARP processors if faults are handled either at slot level or at slot- and register-level.

17.2% of the processors fail due to faults in the control logic. 81.45% of the processors have no fault in the control logic, but faults in the register file. And only 1.35% of the processors have all faults in the data path. Out of these processors, there is not a single VARP processor where all faults are located in a single slot. 0.01% of the VARP processors have all faults in exactly 2 slots, and

0.34% of the VARP processors have all faults in three slots such that there is only one working slot left. These are together the 0.35% operational processors.

The results of the same fault injection experiment are shown in the lower part of table 3-9, when rescheduling is done for fault handling at slot level together with register renaming. In this scenario, a system is operational, when all faults are located in the register file or in at most three slots. Again, 17.2% of the processors fail due to faults in the control logic. Only 0.001% of the VARP processors have all 10 faults in the register file, i.e., all slots are faultless. 33.93% of the processors have four faulty slots. Hence, they are not operational. In total a considerable amount of 44.58% of the VARP processors have at least one working slot left. These are the operational VARP processors. This proofs that the number of surviving VARP processor is increased significantly, if faults are handled in a coarse grained manner at slot level in combination with register renaming.

Finally it is shown that many faulty slots of the VARP processor in the fault injection experiment do not have catastrophic faults. This means, by applying the repair routine at bypass and read port level, many of these faulty slots can be used for some computations. Whether or not an operation can be executed in such a slot depends at least on the fault state of the read ports, bypasses, and execution units in these slots. The following simplifications are used for determining whether a VARP processor is operational or not:

- It is assumed that each execution unit supports only two operation types. An execution unit is considered as operational, if at least one of its operators is faultless.
- About 60% of the cell area of a bypass is occupied by the multiplexer trees of the bypass. The remaining area is occupied by the control logic of the bypass. A bypass is considered as operational, if its control logic is faultless.
- A slot is considered as operational, when all pipeline registers are faultless and both bypasses and the execution unit are operational. It is assumed that faults in the read ports do not cause a catastrophic fault, because such a fault situation can be handled at bypass level.
- Moreover, at least one operator of each type must be located in an operational slot.

Using these assumptions, the slots that were considered as faulty in table 3-9 were reinvestigated. Table 3-10 shows for each VARP processor that has faults in exactly k slots after the fault injection experiment, how many of these slots are affected by a catastrophic fault, and how many of these slots are still considered as operational.

Referring to the lower part of table 3-9: Systems that have faults in exactly k slots	Percentage of VARP models where j out of k slots have a catastrophic defect				
	j=0	j=1	j=2	j=3	j=4
$k = 1$	0.12%	0.16%	-	-	-
$k = 2$	2.0%	3.7%	1.7%	-	-
$k = 3$	7.0%	15.9%	11.3%	2.6%	-
$k = 4$	6.0%	13.9%	11.2%	2.8%	0%

Table 3-10: Percentage of the systems with catastrophic faults in j slots.

For example, according to the lower part of table 3-9, 7.36% of the processors have faults in exactly $k = 2$ slots. In row $k = 2$ of table 3-10 it is further classified how many VARP processors with faults in two slots have catastrophic faults in j slots. For example, 2.0% of the VARP processors have $k = 2$ faulty slots and catastrophic faults in $j = 0$ slots. I.e., both slots can be used for executing some operations. Only 1.7% of the VARP processors with two faulty slots have catastrophic faults in both slots. In 3.7% of the processors one out of the two faulty slots has a catastrophic fault. Finally, for $k = 4$ only 0% of all processors have a catastrophic fault in all four slots (see last column of table 3-10). However, 4.4% of the VARP processors have at least one operational slot, but in all four slots the same operator is faulty. I.e., these processors are considered as non-operational. Together with the 17.2% of the processors that have a fault in the control logic only about 21.6% of the VARP processors did not survive, if the fine-grained self-repair method is used. Thus, 78.4% instead of 0.35% survived, when using a fine-grained self-repair method. I.e., if a population of VARP processors with 10 faults injected in each of them is considered, then the amount of surviving processors is increased significantly by using a fine-grained self-repair scheme with register renaming.

From the results of the fault injection experiment it can be also derived, how many registers of the register file should be preserved as backup registers. Obviously 10 backup registers will be sufficient for handling 10 injected faults. However, table 3-11 shows that it will be sufficient to have only 6 backup registers, because only about 1.5% of the VARP processors from the experiment have defects in more than 6 registers after fault injection.

Faulty registers	1	2	3	4	5	6	7	8	9	10
VARP processors in %	9.0	20.4	26.9	22.7	12.8	4.9	1.3	0.2	0.02	0.0008

Table 3-11: Percentage of the VARP processors, where a particular number of registers is faulty after the fault injection experiment.

3.4 Limitations of a Software-Based Self-Repair Approach

A limitation of the presented fine-grained software-based rescheduling arises from the combination of particular faults in all slots. For example, suppose an *add*-operation of the user application uses registers *r1* and *r2* as source operands. Furthermore, half of the slots cannot access register *r1* through their left read port, and the other half of the slots cannot access register *r2* through their right read port. Then all slots that have a functioning adder cannot access *r1* and *r2* in that combination. Hence, the *add*-operation cannot be executed from the VARP processor, except the values of *r1* and *r2* may be provided through the bypass. Hence, whether or not an application can be executed on a VARP processor depends on the fault state of the processor and on the operations used in the application. Therefore, the results of the fault injection experiment shown in table 3-10 are best-case results, because some processors considered as operational in that experiment may be not able to execute operations with particular register operand combinations. This limitation may be mitigated by improving the rescheduling algorithm. Consider again the example of the *add*-operation. The values of *r1* and *r2* may be accessed through the bypass. But this is only possible, when the *add*-operation is scheduled within the next two clock cycles after generating the values of *r1* and *r2*. But such considerations are not taken into account from the simple rescheduling heuristic in listing 3-5. As a consequence, the rescheduling algorithm may fail to adapt the schedule to the current fault state of the processor, although such a schedule will exist. For this reason, many improvements of the presented rescheduling-algorithm are possible. For example,

- the register allocation may be changed locally for particular operations,
- more sophisticated look-ahead information may be used during the rescheduling, or
- backtracking can be allowed in the rescheduling algorithm.

In all cases the complexity of the rescheduling algorithm is increased. For this reason such extensions must be used carefully as it was already discussed at the end of section 3.3.4.1. Otherwise there is a significant risk that the benefit of the improved rescheduling does not outweigh the runtime overhead for adapting the user application.

However, there are also faults that cannot be handled with software-based approaches, no matter how much the self-repair routine is improved. A major limitation of both software-based self-repair approaches is that faulty components are not physically decoupled from the rest of the system. Particular physical faults, for example a short between supply voltage Vdd and ground in a processor component, cannot be handled. Moreover, because the physical decoupling is

missing in a software-based approach, only benign faults can be handled. These are faults in a particular component that do not cause a misbehaviour, such that other faultless components are affected in the system by this misbehaviour. In particular, when using software-based approaches, the usage of a faulty slot is avoided by executing NOPs in that slot. But faults in the pipeline registers of a slot may change the opcode of a NOP into the opcode of another operation that disturbs the execution of operations in other faultless slots. This may happen by

- disturbing the control flow, when the opcode of a NOP is changed into the opcode of a branch operation, or by
- disturbing the forwarding network and/or the registers in the register file, when the result of the execution unit in the faulty slot is accidently considered as valid. This will happen, when the opcode of a NOP is changed into the opcode of an ALU operation. If the bit fields of the destination register number in the pipeline registers are not affected by a fault, then the faulty slot will produce only results for register *r0*.

These effects can be mitigated by detecting a NOP already in the fetch stage and generating a load-flag there that is propagated through the pipeline registers, as it is done in the VARP processor. Because the load-flag is a single bit only, the probability that this bit is affected by a fault is lower than the probability that an 8 bit opcode is affected by a fault. Nevertheless, the load-flags generated by the slots represent single points of failures, because there is still a chance that a faulty load-flag disturbs the execution of operations in other slots. This makes faults in the load-flags equivalent to faults in the control logic of the VARP processor, which is also a single point of failure. In both cases, the software-based self-repair does not provide a fault handling mechanism. However, as it was shown by the reliability estimations, the control logic is small enough, such that the provided self-repair capability for the data path increases the reliability of a VARP processor significantly in spite of a non-fault tolerant control logic. However, if needed, also the control logic can be made fault tolerant, but this requires hardware administrated hardware redundancy. For example, a simple TMR-approach can be used, where the full control logic, including the PC, MAR, and MBR, is tripled. This increases the size of the VARP processor approximately by 4%, because the control logic makes up only 2% of the size of the VARP core. The advantage of a TMR-scheme is that no diagnostic test for the control logic is needed. However, other active hardware redundancy schemes, like the one proposed in [91], may be used, too.

3.5 Summary

In this chapter the concept of software-based self-repair for statically scheduled superscalar processors was introduced. The concept was demonstrated for the VARP processor, but it can be used for any statically scheduled processor with redundant data path resources. Basically the presented software-based self-repair techniques can be considered as software-administrated hardware redundancy methods. The benefit of the software-based method is that no hardware extensions are needed in the processor core itself for administrating the redundant hardware. This maintains the simplicity of the data- and control path of statically scheduled processors. Hardware-based reconfiguration schemes, as considered in chapter 2, will have a strong impact on the simplicity of embedded processors, because rebinding of operations must be organized during the execution of the user application, which may require a complex hardware-based administration. By using software-based methods, this complexity can be shifted to software. This reduces the hardware overhead, which in turn reduces the vulnerability of the processor core to temporary faults, too. Moreover, because the core itself does not need any hardware extensions, a software-based approach can be used for making processors off the shelf fault tolerant that have not included fault tolerance techniques. For Harvard-architectures that do not have access to their program memory, a work-around was presented by introducing an arbiter. Two strategies for adapting the user application in the field to the current fault state of the processor during the startup phase were presented: software-based rebinding and software-based rescheduling.

The advantage of the software-based rebinding is that the adaption can be done locally in each instruction. This does not change the length of a basic block, and complex operations for relocating basic blocks and patching branch-operations are not needed. On the other hand, multi-cycle operations cannot be handled. Moreover, by the fault tolerant list-scheduling, presented in listing 3-4, the performance of the application is degraded statically during the development phase. For this reason the performance is not further degraded in the field, if the application is adapted to a fault situation. This allows for well predictable real time behaviour.

The software-based rescheduling can handle multi-cycle operations but at the cost of a more complex administration, because the schedule length of basic blocks may be increased. On the other hand, the performance of the user application is only degraded as much as necessary for handling the current fault state. Thus, fault handling in the field may cause graceful performance degradation. Moreover it was shown that the granularity for fault handling can be lowered down to multiplexer level for read port and bypass structures in the VARP processor, without

increasing very much the complexity of the rescheduling algorithm. This avoids a strong graceful performance degradation, when faults in single and multiple components must be handled in the processor, because many components that are affected by a fault can still be used partially for executing particular operations. However, similar to the hardware-based approach, handling of multiple faults will increase the reliability only significantly, if the processor is affected by high failure rates. In particular the fine-grained self-repair should not be considered as a technique for improving the reliability. But it is very useful for avoiding strong performance degradations after adapting the user application to the current fault state of the processor.

Both software-based approaches can handle only permanent faults that are detected during the startup test or by a test in idle-times. Extra methods for handling transient faults and permanent faults that occur during the runtime of the user application must be included in the system. Furthermore, for very large applications, the adaptation of the user application may take a few seconds during the startup. The next chapter provides solutions for these two drawbacks by considering hybrid approaches that combine hardware-based and software-based administration schemes.

Chapter 4

Hybrid Self-Repair Techniques

This chapter introduces hybrid self-repair approaches [160, 165]. These are approaches where the administration of hardware redundancy takes place at software- and hardware-level. Such approaches are also referred to as cross-layer approaches [48]. The two hybrid approaches presented in this chapter are obtained by a combination of the methods presented in chapter 2 and chapter 3. A major draw-back of the hardware-based self-repair presented in chapter 2 is the performance degradation during the runtime of the user application, because operations are allocated dynamically to faultless slots. Major draw-backs of the software-based approaches from chapter 3 are the possibly long repair-time during the startup process and that only permanent faults can be handled, which are detected during the startup phase. By the combination, the hardware-administration of the self-repair process helps to overcome some limitations of the software-administration and vice versa. The first hybrid approach presented in section 4.2 reduces the repair-time during the startup phase of the system. Furthermore, the runtime overhead caused by the hardware-based rebinding is reduced by applying software-based self-repair methods. The second hybrid approach presented in section 4.3 allows for handling permanent and transient faults on-line in a coarse-grained manner and performs a fine-grained self-repair off-line. For this reason a coarse-grained on-line self-test is introduced that allows for detecting permanent and temporary faults by concurrent checking.

4.1 Related Work

Recently, cross layer approaches gained more interest for building fault tolerant systems [33, 48, 50, 79, 119]. Most of the fault tolerance approaches presented and discussed so far work only on a single administration layer of the system stack. The layers of the system stack were shown in figure 1-9. According to [33] the draw-back of a single layer approach is that it must hide almost all faults from the layers above, such that these layers can assume a faultless system. This simplifies the implementation of the higher layers, but in most cases this introduces very much overhead in the single layer that has to provide fault tolerance methods for a broad variety of faults. This wastes resources, because fault tolerance techniques for some rare situations may be better implemented at higher layers, for example in software. For this reason techniques are proposed where the detection, diagnosis and fault handling takes place at different system layers. For example, diagnosis and reconfiguration are considered in [33] as hardware-specific tasks[21], while recovery can benefit very much from being executed in software. Such a scheme is proposed [7]. There a combination of time redundancy managed at the operating system layer and error detection codes employed in the hardware-layer is used. The operating system manages the execution of each task at least two times. The results of both tasks are compared. If they are different, then the task is executed a third time for performing a majority vote. However, the hardware also supports error detection by error detection codes, but no error recovery. When the hardware detects an error during the execution of a task, then the operating system is notified. The operating system immediately terminates the running task and starts the task again. Hence, error recovery is done at software-level, and runtime is saved by early termination of running tasks when an error is detected by hardware layer methods. In [145] a combination of hardware- and software-based techniques for handling SEUs is proposed, whereby hardware-based techniques are only used for supplementing fault handling in processor components for which no software-implemented hardware fault tolerance method [144] exists. An even tighter coupling of administration layers is claimed in [33]. Applications may be also aware of unreliable hardware, by providing interfaces for activating or deactivating particular features. For example, when the operating system becomes aware of some faults in the processor, then some features of the applications can be deactivated through such an interface, instead of terminating and restarting the whole application.

[21] In that way the software-based self-repair techniques provide an interesting supplementation for reconfiguration methods used in cross-layer approaches, because these approaches allow for reconfiguration at software-layer.

The tight integration of different administration layers also influences the compiler. Compilers can play an important role for cross-layer approaches. The already presented methods from Bolchini et al. [24] and Holm and Banerjee [82] employ the compiler for generating self-checking code by generating redundant computations and compare-operations. However, in both methods the redundancy is introduced in a very unspecific way by duplicating most of the operations of an application. In [55] the problem of selective protection of particular pieces of source code is considered. For this reason the data types in the source code can be annotated as reliable or unreliable. With these annotations the compiler can generate code that uses information and/or time redundancy for making such variables and operations on these variables more reliable. Hu et al. presented in [83] a hybrid approach, where the compiler is aware of some fault tolerance feature of the processor. Operations are duplicated by the compiler and comparison of the results is done in hardware. For this reason, a special register queue was added to the data path as well as some control bits to each operation. These control bits are set by the compiler and determine whether an operation is an original one (and must write its result into the register queue) or a duplicated one, whose result must be compared with the result of the original operation, which can be found in the register queue. A recovery scheme is not included in this work. Such a hybrid approach provides a compromise between runtime overhead and flexibility. On the one hand the runtime overhead is reduced compared with methods that perform comparison of results in software. On the other hand, the flexibility of duplicating only selected operations is maintained.

However, administration schemes in cross-layer approaches must not span from hardware layers to software layers. Cross-layer approaches may also work only at different hardware administration layers. For example, various fault tolerance methods detect temporary faults locally, e.g., the RAZOR-Flip-Flop detects delay faults [56] in registers and an improved scheme in [173] detects also temporary faults in the combinatorial components writing into registers. For error correction time redundancy is used, i.e., an additional clock-cycle is needed, which changes the timing in the sequential system. For this reason an additional administration at system-level is needed that controls the global timing. This administration scheme may be a global stall signal routed to all register elements of the system, which may cause a strong overhead in wiring. But even more complex administration schemes are implemented in hardware. Shyam et al. presented a cross-layer approach in [169] that works at system- and register-transfer level. Computations in a processor pipeline are divided in computation periods. Before a computation period starts, a checkpoint is taken at micro-architectural level. During the computational period, which takes typically some hundreds of clock cycles, pipeline components are tested at register transfer level, when they are in

idle mode. When the test detects a fault, the data path is reconfigured and a roll-back to the latest checkpoint is done.

4.2 Hybrid Off-Line Approach

The hybrid approach presented in this section works at hardware- and software-level. Moreover, both methods do not need to communicate with each other. The configuration of the system that implements the hybrid off-line self-repair approach is shown in figure 4-1.

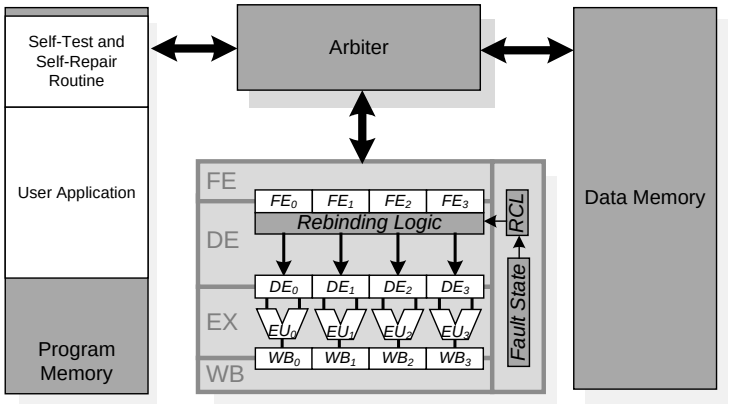


Figure 4-1: VARP system that implements the hybrid off-line self-repair approach.

In this configuration one of the software-based self-repair routines from chapter 3 is executed on the fault tolerant VARP processor that supports hardware-based rebinding (see section 2.3). For this reason the system also contains the arbiter. The self-repair routine is stored in the program memory. The fault tolerant VARP processor has a rebinding logic that is placed at the output of the fetch register (see also figure 2-7). The rebinding logic allows for handling of multiple permanent faults by rebinding operations on-line to other slots, and it can be used for fault handling at slot level or at execution unit level. The rebinding logic is controlled by a rebinding control logic (RCL) that has access to the fault state of the processor. Please recall that the fault state at execution unit level is given by the fault state registers that correspond directly with the fault state functions *fsSlot* and *fsEU* specified in section 2.2.1. Moreover, the same fault state registers are accessible with the VARP-operation *ld_fs y,z,r* that reads bit *z* of fault state register *y* and stores this value in the processor register *r* (see Appendix A). Hence, the hardware-based rebinding and the software-based self-repair routine have access to the same fault state.

Depending on the software-based self-repair method that is stored in the program memory, the hardware-based rebinding is combined either with the software-based rebinding or with the software-based rescheduling. In the letter case the hybrid method is denoted as *hardware-supported rescheduling* (*HSRS*) and in the former

case the hybrid method is denoted as *hardware-supported rebinding* (*HSRB*). In both cases the hardware-based rebinding as well as the used software-based method can be used both for adapting the processor to the current fault state. This combination is used for

- simplifying the startup process of such a system,
- using the hardware-based rebinding as backup repair process for the software-based method,
- performing only a partial adaptation of the user application.

Please note that the combination of software-based rescheduling with hardware-based rebinding is only considered for the coarse-grained version of the rescheduling, i.e., fault handling takes place at slot and execution unit level. By this, both methods can use the same fault state information. The three mentioned options from above are described in more detail in the subsequent sections.

4.2.1 Simplified Startup Process

During the startup of the hybrid system, a diagnostic self-test must be performed in the same way as it was done for the software-based approach. This requires that the rebinding logic is deactivated; i.e., each operation of the self-test program is executed in that slot to which it was statically scheduled. Otherwise, faults may be detected in the wrong slot. The fault state of the data path is stored in the fault state registers. Then the rebinding logic is activated and either the software-based rebinding or the software-based rescheduling is executed. In contrast to the startup process that is used for the software-based method, it is sufficient to have a single version of the self-repair routine. Multiple versions of the self-repair routine are not needed, because the hardware-based reconfiguration adapts dynamically the self-repair routine to the current fault state. Moreover, the restriction that at least a single slot must be faultless can be discarded. I.e., operator faults are allowed in all slots, because the hardware-based rebinding maps each operation into a slot where it can be executed.

Thus the hardware-based self-repair is used for fault handling during the execution of the self-repair routine, and the software-based self-repair routine is used for fault handling in the user application. The performance degradation caused by the hardware-based self-repair routine during the execution of the software-based self-repair routine is negligible, because it just affects the execution time of the self-repair routine, but not the execution time of the user application. Please note that it is not necessary to deactivate the hardware-based rebinding during the execution of the user application. The user application has been adapted by the software-based self repair according to the fault state that is stored in the fault state

registers. The same fault state is used by the hardware-based rebinding. Thus, according to the given fault state, no operation uses a defect component, and therefore the hardware-based rebinding will not take any action during the execution of the user application.

4.2.2 Hardware-based Backup Repair

The next benefit of such a hybrid method comes from the fact that software-based methods are statically limited in their fault handling capability. In particular, fault handling of the software-based rebinding is limited to the number of operator- and execution unit faults that were specified for the generation of the fault tolerant schedule. The software-based rescheduling is limited by the available backup program memory for the prolonged schedules. Hardware-based rebinding helps to overcome these limitations. The details are discussed separately for both software-based approaches.

4.2.2.1 Hardware-Supported Rebinding

When the number of specified operator- or execution unit faults is exceeded by the current fault state, the rebinding algorithm may fail for some instructions. However, it will succeed for most other instructions. An example is given in figure 4-2 (a).

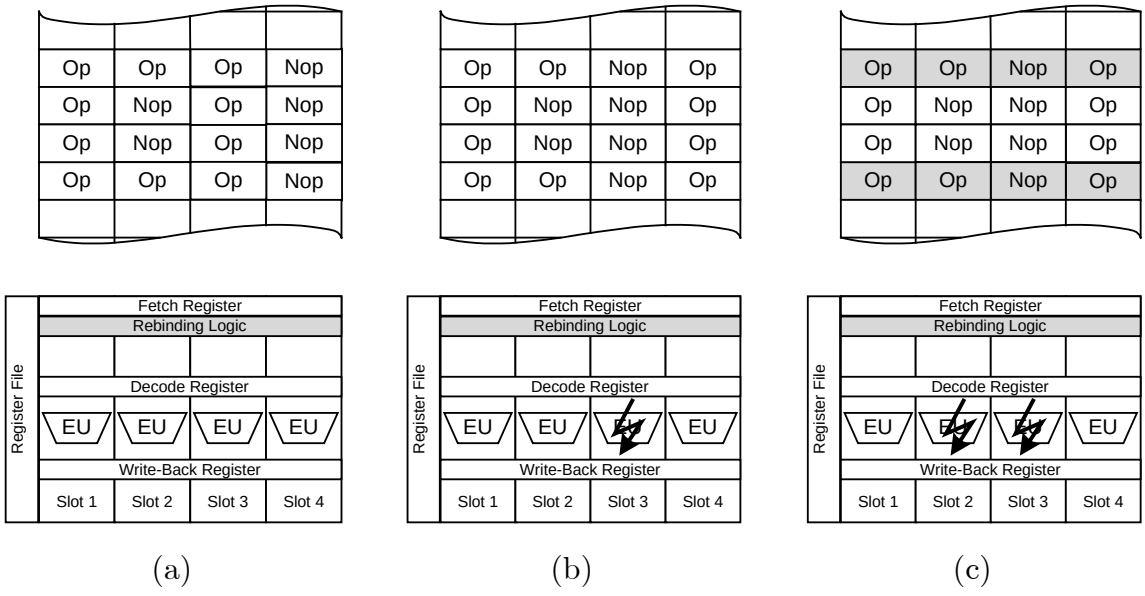


Figure 4-2: (a) VARP processor with fault tolerant schedule. (b) Fault tolerant schedule from (a) is adapted to fault in slot 3. (c) Fault tolerant schedule from (a) cannot be adapted to faults in slot 2 and 3. Fault handling for gray instructions is overtaken by the rebinding logic.

Suppose the schedule has been generated with a fault tolerant list scheduling algorithm for tolerating a 1-slot-fault. Thus each instruction contains at least a single NOP. But some instructions contain more NOPs, because there was no more

instruction-level parallelism available in the user application. Furthermore, suppose the VARP processor shown in the lower part of figure 4-2 (b) has a fault in slot 3, then each instruction can be adapted successfully by the software-based rebinding algorithm, as it is shown in the upper part of figure 4-2 (b), because the specified number of faults is not exceeded by the currently present number of faults in the processor. Now suppose that a second fault occurs in slot 2, as shown in figure 4-2 (c). Then the gray shades instructions shown in the upper part of figure 4-2 (c) cannot be adapted to this fault situation, while the white instructions can. In this situation the software-based rebinding for the user application will fail. But a failure of the system may be avoided, if the hardware-based rebinding is used as backup repair routine. I.e., those instructions that cannot be adapted with the software-based rebinding are left unchanged. If such an instruction is fetched in the processor, the hardware-based rebinding will detect that this instruction cannot be executed, and it will be adapted dynamically to the current fault state.

The white instructions will be executed by the fault tolerant VARP processor without any delay penalty, because they were adapted by the software-based rebinding such that they already contain NOP operations for each faulty slots. As a consequence, the execution speed of the user application is only degraded for those instructions that could not be adapted by the software-based rebinding. Hence, during the generation of the fault tolerant schedule, it becomes possible to trade the length of the fault tolerant schedule against the runtime overhead caused by the hardware-based rebinding. From another point of view this combination of the software-based and hardware-based rebinding can be considered as a performance improvement of the hardware-based rebinding. For example, suppose the original schedule has been generated in such a way that it is not fault tolerant. Nevertheless, there are many instructions that can be adapted from the software-based rebinding to an occurred fault situation. By adapting these instructions permanently in the program memory they must not be adapted from the hardware-based rebinding, which, in turn will not increase the execution time of these instructions.

4.2.2.2 Hardware-Supported Rescheduling

The static limitation of the software-based rescheduling arises from the available backup area in the program memory that is needed for storing instructions of the prolonged schedules. When this backup area is exhausted, then further adaptation of the user application will be only successful, if this adaptation does not extend the length of the schedule. When the hardware-based rebinding is available as backup-repair method, the software-based rescheduling can leave basic blocks unchanged, whose adaptation to the current fault state will extend their length.

4.2.3 Partial Adaptation

The partial adaptation is a more systematic way of using the hardware-based rebinding as a backup-repair mechanism. When partial adaptation is used, then only some specified regions of the user application are processed from the software-based self-repair methods, before the user application is launched. For the other regions that were not processed, the fault handling is done completely by the hardware-based rebinding. Moreover, even for the processed regions, the hardware-based rebinding can be used as backup-repair technique, when the software-based method was not successful. The partial adaptation can be used for two purposes:

- Reducing the startup time.
- Adapting the user application during short execution breaks of the user application.

4.2.3.1 Reducing the Startup Time

In order to reduce the startup time, the user application may be only adapted partially to the current fault state. For this purpose, the user application is partitioned into critical and non-critical code sections. Critical code sections are, for example, time-critical sections in real-time applications. Only the critical code sections will be adapted with the software-based methods to the current fault state during the startup of the system. Therefore, the instructions in those critical code sections must be not adapted dynamically with the hardware-based method, and therefore they can be executed without any delay penalty introduced from the hardware rebinding.

Non-critical code sections are not adapted with software-based methods during the startup of the system. Thus, instructions in these sections will be adapted dynamically by the hardware-based method during the execution of the user application. This, in turn, creates a delay penalty. However, when the partition of critical and non-critical section is done in an appropriate way, then delay penalty will not affect the real-time constraints of the user application.

4.2.3.2 Adaptation in Execution Breaks

Partial adaptation can be also used for adapting the full user application piecewise during execution breaks. This may be useful for real-time systems that have to provide the required service in fixed time intervals, but short activity breaks are available that can be used for periodically testing or self-repairing. The utilization profile of such a system is shown in figure 4-3.

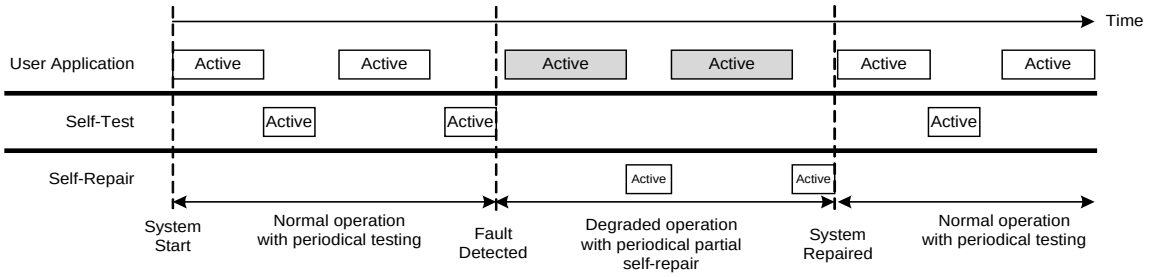


Figure 4-3: Utilization Profile for the cross-layer Approach. Gray shaded activities of the user application denote emergency operation mode.

In this scenario a self-test methodology must be available that can be used for periodically testing of the system. When the test will detect a permanent fault, then the fault state is stored in the fault state memory, and the execution of the user application is resumed without performing a time-consuming software-based self-repair of the user application. In the example in figure 4-3 this happens after the second activity phase of the user application. During the third and fourth activity phase of the user application, the hardware-based rebinding is used for adapting dynamically the user application to the current fault state. This increases the runtime of the user application due to the introduced rebinding cycles. This is denoted by the prolonged gray bars in figure 4-3. In turn, this will shorten the execution breaks. This state of the system can be considered as an *emergency operation mode*, where the service of the user application is still provided, but with reduced performance. However, during the shortened execution breaks, a software-based self-repair approach is activated in order to adapt partially the user application. Each activation of the software-based self-repair routine adapts another part of the user application. After adapting the complete application, the self-test is periodically activated again in order to detect further faults. For this scenario only the HSRB is well suited, because it allows for a local adaptation of the user application without relocation of basic blocks and branch operation patching.

4.2.4 Results

The benefit of the proposed hybrid method regarding runtime overhead and reliability is quantified in this section.

4.2.4.1 Runtime Overhead

The runtime overhead can be divided into a *static* and a *dynamic* part. The static part is originated by the software-based self-repair methods during the system startup, when the user application must be adapted to the current fault state. This static overhead can be reduced with the proposed hybrid methods, when they are used for partial adaptation. However, the partial adaptation increases the dynamic

runtime overhead, because for some sections of the user application the hardware-based rebinding is used as backup-repair mechanism. In order to quantify these impacts the following equation is used for estimating the dynamic runtime t_{dyn} of some user application.

$$t_{dyn} = \frac{l \cdot cr \cdot exe_{cr} \cdot (1 + dynOvh_{cr}) + l \cdot (1 - cr) \cdot exe_{ncr} \cdot (1 + dynOvh_{ncr})}{clk} \qquad (4\text{-}1)$$

In that equation l is the length of the user application in instructions. cr is the fraction of instructions that belong to the time critical part of the application. exe_{cr} is the execution frequency of the time critical part and exe_{ncr} is the execution frequency of the non-time critical part of the user application. Typically less than 10% of the code in signal processing applications is time critical and more than 90% of the execution time is spend for executing these time critical parts [138]. $dynOvh_{cr}$ is the expected runtime overhead for the time critical part of the application after adapting that part to a single fault. Accordingly, $dynOvh_{ncr}$ is the expected overhead for the non-time critical part. According to the results in table 3-6 it is assumed for 1-slot-faults that software-based rebinding creates on average a dynamic runtime overhead of 30%, software-based rescheduling creates on average a dynamic runtime overhead of 25%, and hardware-based rebinding creates on average a dynamic runtime overhead of 99%. These properties are used for estimating the dynamic runtime t_{dyn} of some applications that differ in length l with equation (4-1). I.e., for these applications it is assumed that they are composed of basic blocks that have the same properties as the benchmarks listed in table 2-4. The results are shown in table 4-1.

Instructions in application	**runtime for faultless processor**	**dynamic runtime for a 1-slot-fault**			
		SWRB $dynOvh_{cr} = 0.30$ $dynOvh_{ncr} = 0.30$	**HSRB** $dynOvh_{cr} = 0.30$ $dynOvh_{ncr} = 0.99$	**SWRS** $dynOvh_{cr} = 0.25$ $dynOvh_{ncr} = 0.25$	**HSRS** $dynOvh_{cr} = 0.25$ $dynOvh_{ncr} = 0.99$
16000	0.0576 msec	0.07 msec	0.09 msec	0.07 msec	0.09 msec
64000	0.2304 msec	0.3 msec	0.38 msec	0.29 msec	0.38 msec
256000	0,9216 msec	1.2 msec	1.5 msec	1.2 msec	1.5 msec
Overhead	**0%**	**30%**	**64%**	**25%**	**62%**

Table 4-1: Estimated dynamic runtime overhead in milliseconds for three different application scenarios.

The runtimes of the applications on a faultless processor are shown in the second column. These runtimes are used as baseline (runtime overhead is 0%). The runtimes of the applications after adaptation to a 1-slot-fault by a software-based or hybrid method are shown in the remaining columns. Please note that the runtime overhead is 99%, if hardware-based rebinding is used (see table 3-6). As already shown in table 3-6 the runtime overhead of software-based rebinding (SWRB) is 30%, and the runtime overhead of software-based rescheduling (SWRS) is 25%. When the applications are adapted by using the hardware-supported

rebinding (HSRB), the runtime overhead increases to 64%, but it is far below the overhead for hardware-based rebinding. A similar result is obtained for the dynamic runtime overhead, if the application is adapting with hardware-supported rescheduling (HSRS). In this case the dynamic runtime is increased by 62%.

This reduction of the dynamic runtime overhead from 99% down to 64% respectively 62% by the hybrid approach is achieved by adapting the time critical part of the user application with software-based methods. On the other hand this increases the static runtime overhead. This static runtime t_{static} is estimated by using the equation

$$t_{static} = \frac{l \cdot cr \cdot tRep_{cr}}{l_{fragment}} + \frac{l \cdot (1 - cr) \cdot tRep_{ncr}}{l_{fragment}}. \tag{4-2}$$

Thereby, l is again the length of the user application, and cr is the fraction of time critical code. $l_{fragment}$ is the length of a code fragment that is composed of the benchmark applications listed in table 3-4. According to table 2-4 the length of such a fragment is 71 instructions. $tRep_{cr}$ respectively $rRep_{ncr}$ are the runtimes of the self-repair routine in seconds for adapting a time critical respectively a non-time fragment. It is assumed that the time critical part and the non-time critical part of the application are composed of basic blocks with similar properties, such that the repair time for these basic blocks is almost equal, i.e., $tRep_{cr} = tRep_{ncr}$. According to table 3-4, the execution of the software-based rebinding as repair routine for a fragment with 71 instructions takes 0.00004224 seconds, assuming a clock rate of 500 MHz. According to table 3-8, the execution of the software-based rescheduling for such a fragment takes 0.00117 seconds. Using these assumptions, the static runtime overhead for user applications with various lengths is shown in table 4-2.

Instructions in application	**static runtime for**			
	SW-rebinding $tRep_{cr}$**=0,04224 msec** $tRep_{ncr}$**=0,04224 msec**	**HSRB** $tRep_{cr}$**=0,04224 msec** $tRep_{ncr}$**=0 msec**	**SW-rescheduling** $tRep_{cr}$**=0,04224 msec** $tRep_{ncr}$**=0,04224 msec**	**HSRS** $tRep_{cr}$**=0,04224 msec** $tRep_{ncr}$**=0 msec**
16000	0.009 sec	0,0009 sec	0,377 sec	0,0377 sec
64000	0.038 sec	0,0038 sec	1,511 sec	0,1511 sec
256000	0.151 sec	0,0151 sec	6,044 sec	0,6044 sec
Total	**100%**	**10%**	**100%**	**10%**

Table 4-2: Estimated static runtime overhead in seconds for three different application lengths.

From equation (4-2) follows for the used assumptions that the static runtime of both hybrid methods (HSRB respectively HSRS) is the fraction cr of the static runtime of the corresponding software-based method (SW-rebinding respectively SW-rescheduling). By this the startup time can be reduced to 10%, which is equal to the fraction of the code that must be adapted. In both cases, the static runtime

savings come at the cost of an increased dynamic runtime, because the hardware-based rebinding must be used as backup-repair for the non-adapted parts of the user application, which increases the dynamic runtime for these application parts. However, these are the non-time critical and less frequently executed parts of the application. Thus, the startup time of the system for a newly detected fault is reduced by almost 90%, while the runtime of the user application is increased by 64% (for HSRB) or 62% (for HSRS) instead of by 28% respectively 25%. I.e., the dynamic runtime is increased only by approximately 40%, if hybrid methods are used instead of pure software-based methods. Thus the hybrid approaches may be used for trading startup time against dynamic runtime.

4.2.4.2 Reliability Analysis

The reliability of the hybrid system is almost the same as the reliability of the VARP processor with hardware-based rebinding. Both systems tolerate the same faults. The only difference is that the arbiter must be taken into account as an additional component that must be faultless in the hybrid system. Thus, the reliability function for the hybrid system is given by

$$R_{hyb}(t) = R_{hwr}(t) \cdot R_{arbiter}(t) = R_{hwr}(t) \cdot e^{-\lambda \cdot t \cdot 1014},$$

where R_{hwr} is the reliability function of the VARP processor with hardware-based rebinding and $R_{arbiter}$ is the reliability function of the arbiter. R_{hyb} is plotted in figure 4-4 together with the reliability functions of a non-fault tolerant VARP processor (R_{NFT4}), a fault tolerant VARP processor with hardware-based rebinding (R_{hwr}), and a fault tolerant VARP processor with software-based self-repair (R_{swa}).

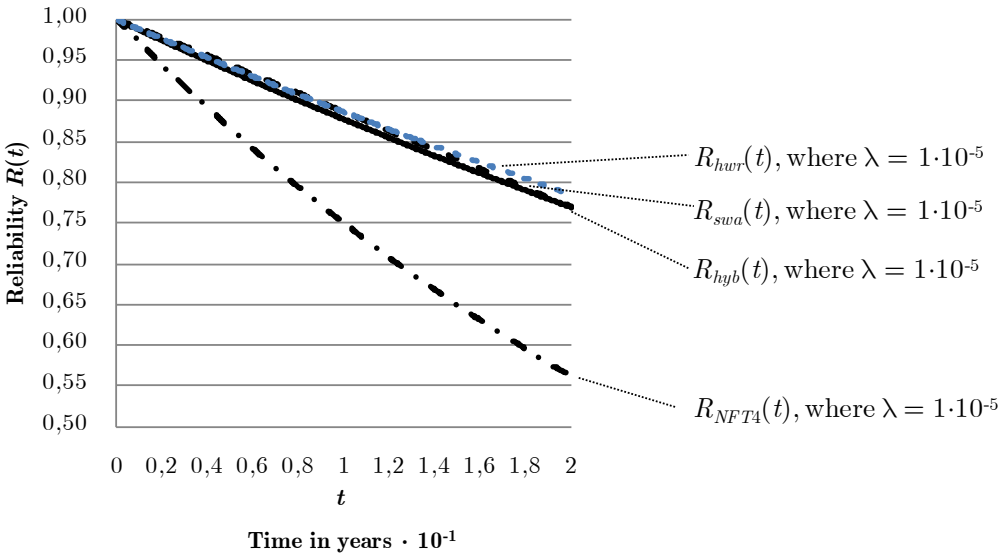


Figure 4-4: Reliability plot of the hybrid system.

R_{hyb} is slightly below the reliability of the pure hardware-based and the pure software-based solutions. When comparing R_{hyb} with R_{hwr}, then the reliability improvement factor for R_{hwr} ranges from 1.07 to 1.08 for the plotted time interval.

4.2.5 Conclusions

The hybrid off-line methods will not significantly change the reliability of an already fault tolerant VARP processor based system. But the hybrid off-line methods can be considered as a way for simplifying the startup process and trading static runtime overhead against dynamic runtime overhead in order to speed up the startup process. By this the static runtime of the software-based rescheduling can be reduced from seconds to some hundreds of milliseconds. This can be an interesting option for large user applications with small time critical code sections.

The granularity of the self-repair is determined by the granularity of the hardware-based rebinding. A combination of a finer grained software-based self-repair with the coarse grained hardware-based rebinding is possible, but requires that the fault states of the processor are represented at different granularity levels, too. For example, a fault in a read port must be represented as a slot-fault for the hardware-based rebinding, but it is not considered as a slot-fault by the finer-grained self-repair scheme.

Table 4-3 summarizes the advantages and disadvantages of the hybrid methods (HSRB and HSRS), software-based methods (SWRB and SWRS) and the hardware-based rebinding (HWRB). This summary can be used to conclude in which situation which self-repair method should be used best.

Method	**Dynamic runtime overhead**	**Static runtime overhead**	**Reliability**	**Hardware Overhead**	**Allowed operation latency**
HWRB	high	**none**	**high**	rebinding logic	1
SWRB	**low**	**low**	**high**	**none/arbiter**	1
SWRS	**low**	high	**high**	**none/arbiter**	**≥ 1**
HSRB	low to medium	**very low**	**high**	none/arbiter + rebinding logic	1
HSRS	low to medium	medium	**high**	none/arbiter + rebinding logic	1

Table 4-3: Summary of the properties of all presented self-repair methods. Bold items are the best ones.

The hardware-based rebinding produces the highest runtime overhead. However, hardware-based rebinding – and hybrid methods, too – provide a slightly higher reliability than the pure software-based methods. When a high dynamic runtime overhead is not acceptable, due to the real-time constraints of the application, then the software-based methods are favorable. Both software-based methods achieve a significant dynamic runtime reduction compared with hardware-based

rebinding, if a fault is present in the system. But there is a significant draw-back of the hybrid methods and the software- and hardware-based rebinding. All of these methods cannot be used for rebinding multi-cycle operations. In this case, only the software-based rescheduling can be used. Another draw-back of both software-based methods, compared with the hardware-based rebinding, is the static runtime overhead originated during the system startup. Depending on the size of the application it may range from a few milliseconds up to a few seconds, if some hundred thousand of instructions must be processed. The hybrid methods provide a good trade-off between the required repair time (i.e., the static runtime overhead) and the dynamic runtime overhead. This makes them attractive for the use in real time applications, where the time critical parts of the executed program are small, and the repair time at the system startup must be short. In this case the static runtime overhead can be reduced significantly, when the software-based methods are applied only to the time critical parts of the user application. Then, these parts can be executed with a low dynamic runtime overhead, which can be important for meeting real time constraints.

Using the presented hybrid methods, the hardware redundancy in the data path of a statically scheduled superscalar processor is handled at two distinct system layers. The rebinding logic is used for administration at the hardware layer, while the software-based methods are used for the administration at application layer. Although the hardware-based rebinding immediately allows after fault detection for avoiding the usage of a defect component, the hybrid method is only suitable for handling permanent faults, because neither concurrent error detection nor error recovery is provided. An extension of the VARP processor that allows for handling temporary and permanent faults on-line is presented in the subsequent section.

4.3 Hybrid On-Line Approach

Now, a hybrid approach is presented that allows for detecting and handling a single transient or permanent fault on-line in the VARP processor. By a combination of this approach with a software-based method, permanent faults in multiple components can be handled, too. The hybrid on-line approach is also a cross-layer approach, because it uses a combination of hardware- and software-based techniques for detecting and handling faults. The redundant operators in the data path are used for concurrent checking by executing each operation twice with distinct operators. Similar to the idea presented in [24], the administration is done in software and in hardware. Each operation in the original user application is duplicated by the compiler such that it can be executed a second time on a distinct execution unit. The results of concurrently executed operations are compared in hardware in order to detect an error. In contrast to the work in [24],

the hardware extension allows also to recover from a detected error within one to two clock cycles and to distinguish between transient and permanent faults. Recovery and fault localization are based on a re-execution of a duplicated operation, if there was a mismatch between the result of the original and the duplicated one. The result of the third execution is used for a majority vote. If a permanent fault was detected, then the fault state is saved such that the results of the faulty unit are masked for the rest of the life time of the system. The hardware extensions of the VARP processor and the needed modifications of the instruction encoding are described in the subsequent sections.

4.3.1 Instruction Encoding

The user application that is executed by the fault tolerant VARP processor assists the processor with the detection and localization of a fault. For this reason the compiler must generate a fault tolerant schedule from the original program by duplicating every operation in the original program. The scheduling algorithm of the compiler must ensure that an *original operation* and its *duplicated operation* are scheduled in different slots. Furthermore, it must ensure that the result of the original operation is not used before the duplicated operation was executed. This restriction ensures that a mismatch between two results is discovered, before a wrong result is used as an input for another operation. Thereby, no complex roll back mechanism is necessary for fault recovery. These restrictions can be incorporated easily in the scheduling algorithm of the compiler. An example of such a fault tolerant schedule is shown in figure 4-5.

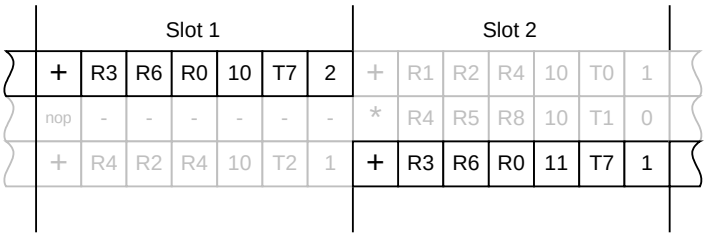


Figure 4-5: Fragment of a fault tolerant schedule with duplicated *add*-operation.

This small code fragment shows the instructions of two slots. The first *add*-operation in slot 1 is the original one, while the second *add*-operation in slot 2 is the duplicated one. For the remaining operations no duplicated operations are shown. Both *add*-operations have the same source and destination registers. I.e., they use R3 and R6 as source and R0 as destination register. Moreover, each operation contains some additional bit fields for encoding some bookkeeping information that was not present in the instruction word of the non-fault tolerant VARP processor. This information is used by the VARP processor for performing the concurrent checking correctly and, if needed, error recovery. The additional information is grouped in the bit-fields *mod*, *RefReg* and *RefFU* of each operation.

Figure 4-6 shows the coding of these bit fields for an operation that is scheduled to slot k.

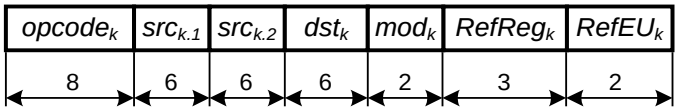


Figure 4-6: Encoding of an operation for issue slot k with bookkeeping information.

The bit fields are filled by the compiler, when an operation is scheduled. Their meaning is as follows:

- $opcode_k$, $src_{k.1}$, $src_{k.2}$, dst_k have the same meaning as for the non-fault tolerant VARP processor. These four sections are pair wise equal for an original operation and its duplicated operation. Consequently, both instructions perform the same operation with the same source operands and write to the same destination register.
- mod_k determines the type of the operation. If $mod_k = 10$, then the operation is an original operation, for which a duplicated operation exists. If $mod_k = 11$, then the operation is a duplicated one (see figure 4-5). Otherwise the operation is an original one, for which no duplicated operation exists.
- $RefReg_k$ is a temporary register number. In the example in figure 4-5 $RefReg_k$ is *T7* for both *add*-operations. If $mod_k = 10$ (i.e., the operation is the original operation), then the result of the operation is stored into the register $RefReg_k$ and into the register dst_k. If $mod_k = 11$ (i.e., the operation is the duplicated one), then the result of the operation is compared with the value stored in $RefReg_k$. By this, a mismatch can be detected between the results of an original and duplicated operation.
- $RefEU_k$ is the number of the execution unit that executes the *reference operation*. If $mod_k = 10$, then $RefEU_k$ is the number of the EU that executes the duplicated operation. If $mod_k = 11$, then $RefEU_k$ is the number of the EU that executes the original operation. This information is necessary to re-execute the duplicated operation on an execution unit that is different from the execution units that executed the original and the duplicated operation. In the example in figure 4-5 the first *add*-operation in slot 1 encodes slot number 2 in the bit-filed $RefEU_1$, and the corresponding duplicated operation in slot 2 encodes slot number 1 in the bit-field $RefEU_2$.

This additional information will support the on-line test and recovery process of the VARP processor. The required hardware extensions of the VARP processor are described in the next section.

4.3.2 Hardware Extensions of the VARP Processor

The required hardware extensions of the VARP processor are shown in figure 4-7.

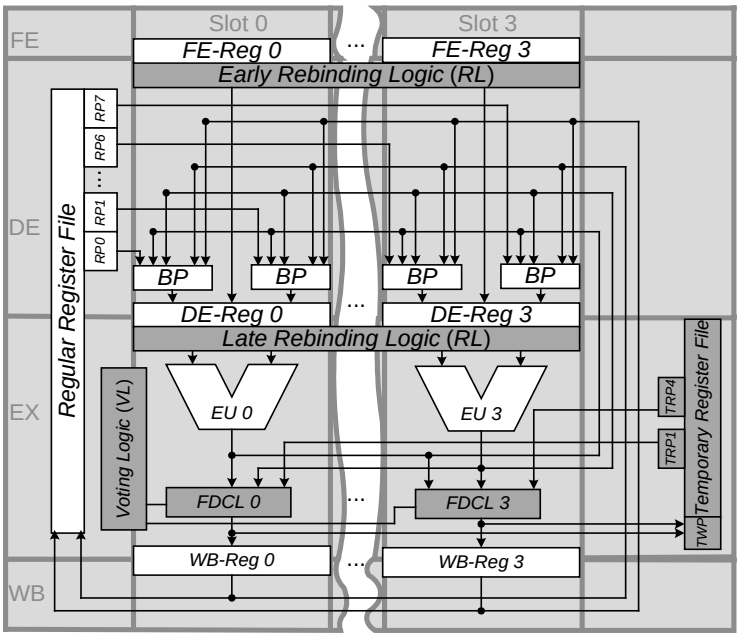


Figure 4-7: Fault tolerant VARP processor with hardware extensions for implementing the hybrid on-line approach.

In order to detect a transient or permanent fault on-line in a slot and to localize the faulty slot, the gray shaded components were added to the non-fault tolerant data path from figure 2-1. These components are:

- A *temporary register file* (*TRF*): The TRF is a collection of a small number of 17-bit registers. These registers store 16-bit values from original operations whose correctness have not been verified so far. Furthermore, a *valid-bit* is saved with each of these values. If this bit is 1, the value is considered as correct. Otherwise the value is considered as wrong, i.e., it was computed in a slot for which it is already known that it contains a permanent fault.
- *Fault detection and compensation logic* (*FDCL*): Each slot k is extended by a FDCL_k that is located between the execution unit EU k and the corresponding write-back register k. This unit has access to the TRF and to the result of the EU k. Thus, it can be used to detect a mismatch between the results of an original operation, which is stored in the TRF, and the result of a duplicated operation, which is generated from EU k.
- A *voting logic* (*VL*): The VL controls the re-execution of a duplicated operation, whose result was not equal to the result of the corresponding original operation.

- A *rebinding logic* (*RL*) that maps an operation that was fetched into slot k, to another slot k' in order to re-execute the operation in slot k'. For the rebinding logic two positions in the data path were considered. Either the rebinding logic is located at the decode stage (*early rebinding* in figure 4-7) or it is located at the input of the execution stage (*late rebinding* in figure 4-7). The position of the rebinding logic affects the fault recovery time as well as the reliability of the system. A discussion on that is given in section 4.3.4. Basically, the rebinding logic has the same structure as the rebinding logic for the hardware-based rebinding already presented in section 2.3. Thus it is used for routing the content of a pipeline register of slot k (either the fetch or the decode register) into any other slot k', while all the other slots are fed with a NOP operation. During normal operation, the rebinding logic for slot k selects the content of the fetch register k. Thus the instructions are executed in those slots to which they were fetched from the program memory.

The rebinding logic contains k fault state registers, such that for each slot k and each operation type t the fault state $fsEU(k,t)$ is accessible.

4.3.3 Concurrent Error Detection

Now it is described how the presented fault tolerant VARP processor detects permanent or transient faults by concurrent error detection. Error detection is done by the fault detection and compensation logics $FDCL_0,\ldots,FDCL_{|SLOTS|-1}$. For this reason FDCL_k receives the bookkeeping information of the currently executed operation in slot k, the result of that operation and the fault state for slot k. Some details of a FDCL are shown in figure 4-8.

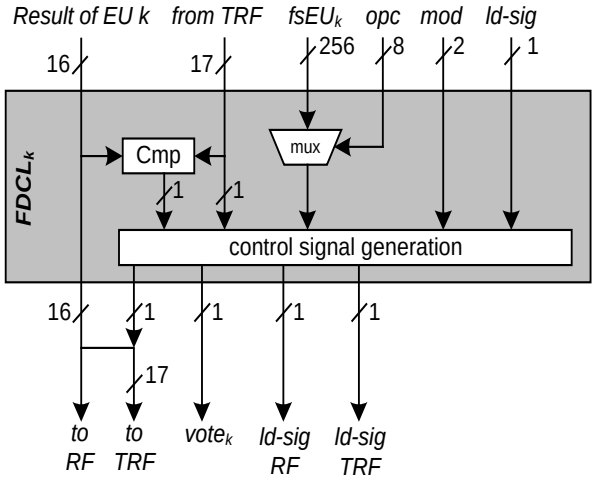


Figure 4-8: Details of the Fault Detection and Compensation Logic (FDCL) in slot *k*.

Two cases must be distinguished. Either an original (case A) or a duplicated operation (case B) is executed. First, case A is considered. It is assumed that an

original operation with type t is executed in slot k; i.e., $mod_k = 10$ for that operation. Depending on the fault state of the processor core, two sub-cases can be distinguished:

1. $fsEU(k,t) = 1$, then the result *res* is considered to be correct and saved into the register dst_k and into the temporary register $RefReg_k$ with the valid-bit set to 1.
2. $fsEU(k,t) = 0$, then the result *res* is considered to be faulty and not saved into register dst_k. But it is saved into register $RefReg_k$ with the valid-bit set to 0.

Thus, after executing the original operation, the result of that operation is stored in the temporary register $RefReg_k$ together with an appropriate valid-bit $valid_k$. Moreover, a result that is obtained from a functioning execution unit (i.e., $fsEU(k,t) = 1$) is stored in the destination register dst_k. Please note that this result may be faulty, because it was not validated yet.

Now case *B* is considered. Either in the same clock cycle or some clock cycles later the duplicated operation is executed in slot k', with $k' \neq k$. A duplicated operation is recognized by its *mod*-bit value ($mod_{k'} = 11$). Furthermore, the duplicated operation encodes the same temporary register $RefReg_k$ as the original operation. The $FDCL_{k'}$ receives the result *ref* of the duplicated operation from EU_k. Furthermore, it receives from the temporary register file the value *res* together with the valid-bit $valid_k$. Both have been stored before in the temporary register $RefReg_k$ by the original operation. Four cases must be distinguished:

1. $fsEU(k',t) = 1$ and $valid_k = 1$, then *res* and *ref* are assumed to be correct and both values are compared by the $FDCL_k$. If $res = ref$, then the original and the duplicated operation have computed the same result and no further action is needed. If $res \neq ref$ then a new fault has been detected and the voting mode of the VARP processor is activated. The voting mode is needed for fault localization. This mode is explained in section 4.3.4.
2. $fsEU(k',t) = 0$ and $valid_k = 1$, then *ref* is considered to be wrong and not saved into the destination register dst_k. Please note that the register dst_k already contains the correct result from the original operation. However, if $res = ref$ holds, then the previously detected fault of operator t in slot k' may was a temporary fault, and $fsEU(k',t)$ is set to 1. With this mechanism temporary faults can be distinguished from permanent faults.
3. $fsEU(k',t) = 1$ and $valid_k = 0$, then *res* is considered to be wrong and the current result *ref* must be stored into the destination register $dst_{k'}$. However, similar to the previous case, temporary faults can be distinguished from

permanent faults by comparing *res* and *ref*. If *res* = *ref*, then the previously detected fault in slot k was a temporary one, and *fsEU*(k,t) is set back to 1.

4. *fsEU*(k',t) = 0 and $valid_k = 0$, then both operators t, in slot k and in slot k', are affected by a fault. Therefore it is assumed that both results *res* and *ref* are wrong. The correct result cannot be determined by a majority voting and a global error signal is set to 1.

The cases listed above are considered in the following example for the *add*-operations in the instruction sequence from figure 4-5. The original operation is the *add*-operation in the first instruction with the source registers *R3* and *R6* and the destination register *R0*. It is executed in slot 1. The duplicated operation is scheduled in the third instruction with the same source and destination registers. It is executed in slot 2. Now suppose that *fsEU*(1,*add*) = *fsEU*(2,*add*) = 1, i.e., the *valid*-bit of the result of the original operation is set to 1 and the reference value of the second operation is considered to be correct, too. The original *add*-operation writes the result into *R0* and together with the *valid*-bit 1 into *T7* (case A.1). The duplicated addition performs the same operation and compares its result with the content of *T7*. If there is no mismatch, then the correct result was already stored into *R0* by the original operation (case B.1). Thus, *R0* contains the verified result. If there is a mismatch, then the voting mode is activated, and a re-execution of the second *add*-operation is initiated. This will delay the execution of the user application. Details of the voting mode are explained in the subsequent section.

Now suppose that *fsEU*(1,*add*) = 0 and *fsEU*(2,*add*) = 1 before the instruction sequence from figure 4-5 is executed. Then the result of the original operation is not written into *R0*, but it is stored together with the *valid*-bit 0 into *T7* (case A.2). When the second *add*-operation is executed, $FDCL_2$ receives the *valid*-bit from *T7*, which is equal to 0. I.e., the content of *T7* is not correct and therefore the result of the second *add*-operation is written to *R0* without verification. It is important to notice that the already detected fault in EU 1 is masked and no delay occurs because no re-execution is initiated. However, this is at the expense of using the result of the second *add*-operation without verification. A second fault that will cause a wrong result of the second *add*-operation will be not detected.

A very similar behavior is obtained for the fault state, where *fsEU*(1,*add*) = 1 and *fsEU*(2,*add*) = 0. The result of the first *add*-operation is treated as correct. Therefore it is written to *R0* and to *T7* (case A.1). When the second *add*-operation is executed, $FDCL_2$ treats the result as faulty, because *fsEU*(2,*add*) = 0. Hence, the result of the second *add*-operation is not written to *R0*, and the result of the first *add*-operation, which has been already written to *R0*, is not verified. The fault in slot 2 is masked, and no delay occurs because no re-execution is done (case B.2).

So far it was explained how an already localized fault is masked during the execution of the user application, and how a fault is detected. In the next subsection it is described how a faulty operator is localized after detecting a mismatch between the both results of an original and a duplicated operation. For this reason the processor is switched into the voting mode. The voting mode is activated by the FDCL that has executed the duplicated operation (see case B.1).

4.3.4 Voting Mode

Fault localization and recovery is based on a second execution of the failed duplicated operation. By this, a third result is obtained that is used for a majority vote, such that the correct result can be selected and the faulty operator is determined. For the second execution of the duplicated operation, the processor is switched into the voting mode. For simplicity it is assumed that the rebinding logic is located at the output of the decode registers. This configuration is referred to as *late rebinding*.

4.3.4.1 Late Rebinding

The voting logic switches the processor into the voting mode, when it receives a *vote*-signal from a FDCL component (see figure 4-9). The voting mode stalls the fetch- and decode-stage of the processor pipeline. By this, the instruction that contains the failing operation stays in the decode register. It is assumed that a mismatch is detected in clock cycle t during the execution of the duplicated operation v' on EU k'. The corresponding original operation v was executed on EU k. This situation is shown in figure 4-9 for the example from figure 4-5. The duplicated *add*-operation is stored in the decode register of slot 2. The result $res =$ 0x0815 is compared with the reference value $ref =$ 0x4711 from $T7$. Because both values are different, the processor is switched into the voting mode by setting the signal $vote_2$ to 1.

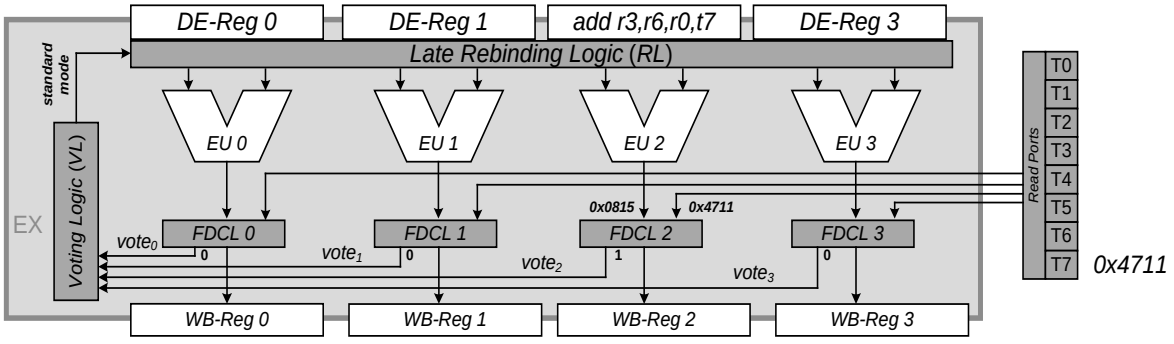


Figure 4-9: Mismatch is detected during the execution of the duplicated *add*-operation from figure 4-5 in slot 2.

The processor switches immediately into the voting mode. I.e., the fetch and decode registers keep their current values. Hence, in the next clock cycle t+1, the

failed *add*-operation is still available in the decode register, such that it can be re-executed. For this purpose, the voting logic determines an execution unit r with $r \neq k$ and $r \neq k'$ that can execute operation v' (i.e. $fsEU(r, type(v')) = 1$). The voting logic sets the control signals of the rebinding logic in such a way that in slot r operation v' is executed in clock cycle t+1, and all other slots execute a NOP-operation during that clock cycle. This situation is shown in figure 4-10.

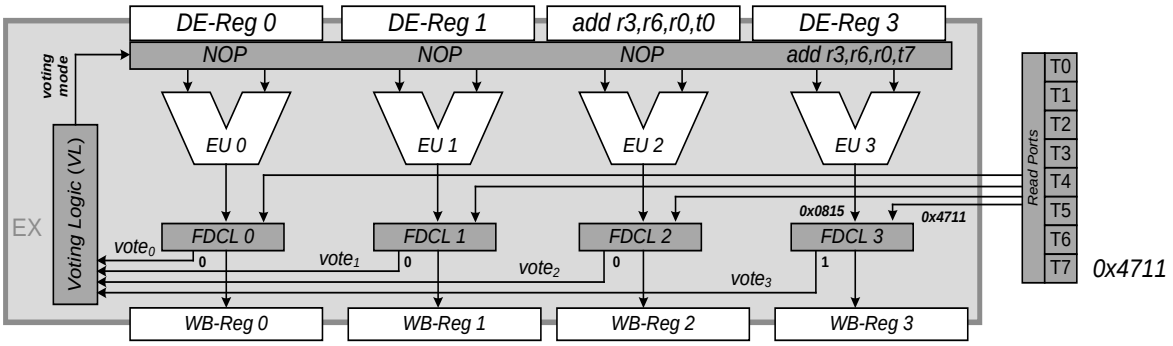


Figure 4-10: Re-execution of the failed *add*-operation from figure 4-9.

The *add*-operation, which resides in decode register 2, is mapped into slot 3 from the rebinding logic. All other execution units execute a NOP. Please note the rebinding logic also maps the *mod*-, *refEU*- and *refReg*-sections of the failed operation v' into slot r. Thus the result of EU r is compared again with the value of $refReg_{k'}$, which was produced by EU k. The outcome of this comparison is observed from the voting logic by reading the signal $vote_r$. According to this signal, the voting logic decides, which execution unit is faulty. If the $FDCL_r$ does not detect a mismatch during the re-execution (i.e., $vote_r = 0$), then the value in $refReg_{k'}$ is equal to the value computed by EU r. Thus, EU k computed a correct value and EU k' is faulty. Therefore, $fsEU(k',t)$ is set to 0. If the $FDCL_r$ detects a mismatch during the re-execution (i.e., $vote_r = 1$), then the value in $refReg_{k'}$ (i.e., the value produced by EU k) was not equal to the result of EU r. Moreover, it was not equal to the result of EU k'. Please note that it is not checked that the result of EU k' and EU r are equal. However, the probability that all three results are different is considered to be very low. Based on this assumption, the conclusion is made that the results of EU k' and EU r are equal. In this case operator t of EU k is declared as faulty. In any case, the result of EU r is considered as correct. For this reason the result is written back into the register file during clock cycle t+2. In clock cycle t+2 the processor is switched back into the normal operation mode. I.e., the operations stalled in the fetch-registers are decoded and the operations of the failing instruction, which reside in the decode registers, are discarded.

A flaw of this approach is that the rebinding logic also maps the operand values of operation v' into slot r. These operand values were loaded during the decode stage from the register file and stored in the decode register. I.e., these values are not loaded again from the register file, when the operation is re-executed. If these

operand values are corrupt, due to a faulty component in the decode stage, for example a faulty bypass or a faulty read port, then the wrong result is generated a second time in slot r. As a consequence an operator in slot k is declared as faulty, although the fault is located in slot k'. This problem is solved by changing the position of the rebinding logic.

4.3.4.2 Early Rebinding

When the early rebinding is used, then the rebinding logic is located at the output of the fetch registers, as it is shown in figure 4-11. In this configuration the rebinding takes place in the decode stage, such that a failed operation uses other bypasses and read ports during its re-execution. Thus, also faults in these components can be detected and handled correctly. However, the administration of the early rebinding becomes a little bit more complex, because a fault is detected in the execution stage, but the failing operation must pass again through the decode stage.

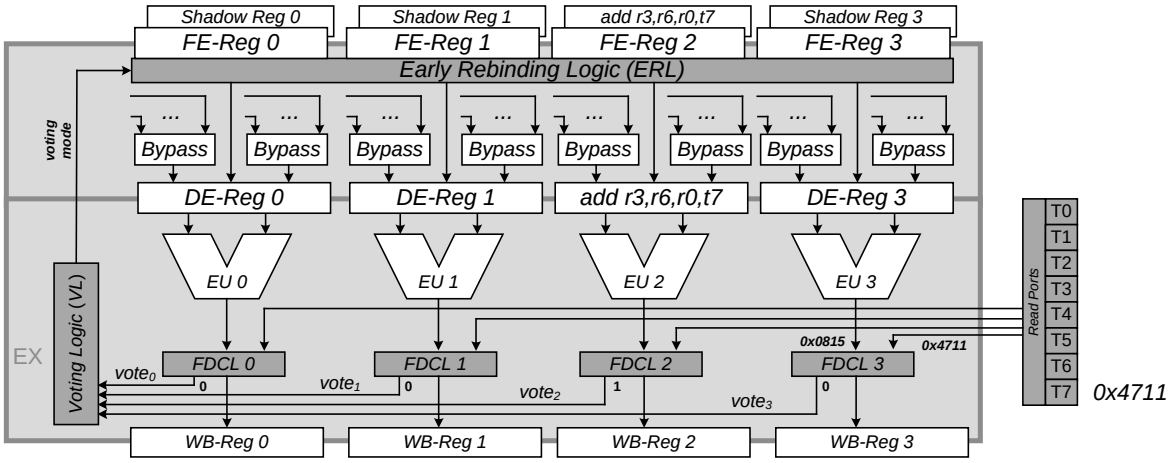


Figure 4-11: Concurrent fault detection with early rebinding and shadow registers.

For this reason a shadow register is introduced for each fetch register. During normal operation, the shadow register of slot k always contains a copy of the operation that is currently in the execution stage of slot k. When the processor is switched in the voting mode, then the fetch registers and the shadow registers maintain their current values. The values generated in the execution stage, except the faulty one, go into the write-back stage. This situation is shown in figure 4-11 again for the example from figure 4-5. For the *add*-operation in the execution stage of slot 2 a mismatch is detected during clock cycle t. A copy of that operation resides in the shadow register of slot 2. In clock cycle t+1 the fetch registers and the shadow registers maintain their current values. The early rebinding logic is controlled by the voting logic in such a way that the *add*-operation in the shadow register is mapped into a slot where it can be executed correctly. Thus, in clock cycle t+1 the *add*-operation is decoded again, and in clock cycle t+2 it is executed

a second time, such that the voting can be done in the same way as for the late rebinding. Moreover, in clock cycle t+2 the operations from the fetch registers are decoded again, such that the execution of this instruction has been delayed by two clock cycles. With this early rebinding more components in the data path are protected against permanent and transient faults.

4.3.5 Limitations

The presented approach allows for the concurrent checking of the results of the execution units. For this reason it can handle all types of operations that generate a result for a register in the register file. This includes memory load-operations, too, because the loaded value from the memory passes the execution unit in the VARP processor. The correct execution of memory store-operations and branch operations cannot be checked. Checking such operations by duplicating them requires more hardware modifications. For example, when duplicating a branch operation, the original branch operation is not allowed to perform the branch; i.e. it is not allowed to write into the program counter. Instead, the branch target must be stored, such that it can be compared with the branch target of the duplicated operation. Signature-based approaches may be used alternatively. Such approaches will check in software, whether or not the program execution follows the correct control flow [71, 129].

Moreover, more sophisticated techniques may be used for distinguishing between transient and permanent faults. The draw-back of the presented approach is that the fault state of an operator may toggle, although the operator has a permanent fault. This is due to the fact that the permanent fault is only activated when particular data is applied. This problem can be solved by using small counters as it was done for example in [170]. Each time a component is detected as faulty, the counter is incremented. When a particular threshold is reached, then the fault state of that component is considered as permanent.

However, the major limitation of the presented hybrid on-line approach is that the fault detection capability gets lost for some operations after the detection of the first permanent fault. This is illustrated in figure 4-12.

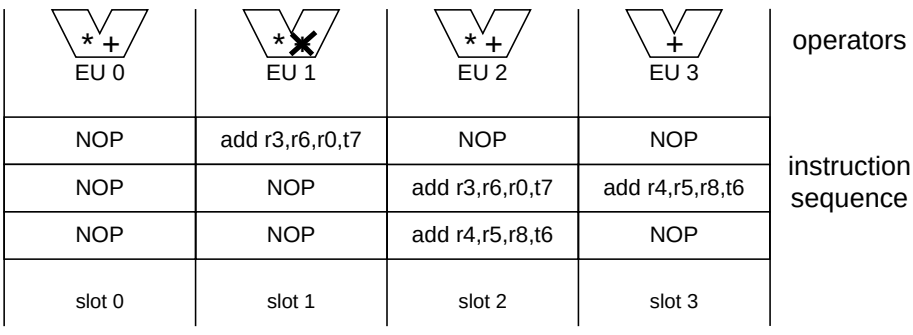


Figure 4-12: Fault detection capability gets lost after the first permanent fault.

Suppose that the *add*-operation in slot 1 is detected as faulty. Then the result of that operation is no longer used, and the correct result is produced from the corresponding reference operation in slot 2. But the result of the reference operation is not checked anymore, such that a faulty value may be used from a dependent operation. Nevertheless, a second fault that affects the *add*-operator in slot 2 can be detected, when another pair of *add*-operations is executed that does not use the same faulty operator pair. In figure 4-12 such a pair of *add*-operations is scheduled to slots 2 and 3. A second fault in slot 2 is detected by this pair of operations. Thus, also the *add*-operator in slot 2 is declared as faulty in this case. As a consequence case B.4 occurs (see page 166), when executing the *add r0,r3,r6,t7* operation in slot 2. I.e., both results are recognized as faulty, but no mechanism is provided for computing the correct result. This problem can be solved by combining the hybrid on-line approach with a software-based self-repair approach.

4.3.6 Combining Hybrid On-Line and Software-Based Approach

The benefits of the combination of the hybrid on-line approach with the software-based self-repair approach are briefly discussed now[22]. The hybrid on-line approach can be combined either with the software-based rebinding or with the software-based rescheduling. Such a system is composed of a VARP processor with the hardware extensions shown in figure 4-7, and a program memory that contains a software-based self-repair routine. The software-based self-repair routine is used for adapting off-line the user application to the fault state that was detected on-line. Figure 4-13 shows how both methods collaborate.

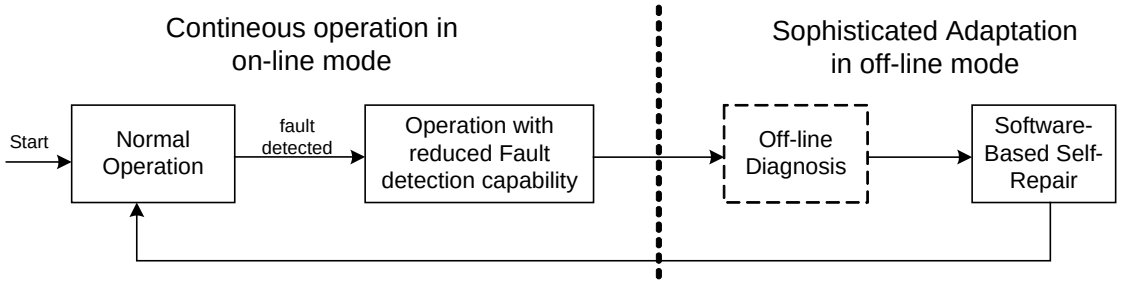


Figure 4-13: System states of a VARP processor that combines hybrid methods with software-based self-repair.

During normal operation all operations are executed by functioning components. Immediately after the concurrent detection of a fault with the hybrid method, the fault is masked with the techniques of the hybrid approach, until there is time for a more sophisticated adaption of the user application. During this time the

[22] In order to simplify the explanation it is assumed that the hybrid on-line approach uses the early rebinding.

execution of the user application is not fully protected against a second fault in the data path of the processor. This is not a strong limitation, if the probability of the occurrence of two consecutive faults within a short time period is very low. A more sophisticated adaptation is done with the software-based methods during execution breaks, or when the system is restarted after some time. Then the processor is in the off-line mode, and the user application is adapted in such a way that the faulty component is not used anymore as it is shown in figure 4-14 for the example from figure 4-12.

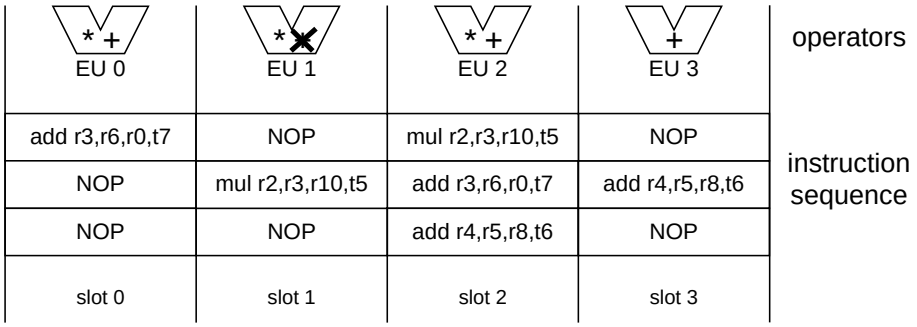


Figure 4-14: Fault detection capability is restored after software-based self-repair.

In the example from figure 4-12 a fault was detected in slot 1. For this reason the *add*-operation that was scheduled to slot 1 is moved to slot 0 and the *mul*-operation from slot 0 was moved to slot 2. After this adaptation, the usage of the faulty operator is avoided, and all results for the register file are concurrently checked. Thus, according to figure 4-13, the system is back to normal operation mode. Please note that the software-based self-repair routines presented in chapter 3 must be slightly modified, because some more dependencies between the operations must be respected.

4.3.6.1 Modifications of the Software-Based Rebinding Algorithm

When the software-based rebinding is used, then data dependencies in the user application are never violated by the rebinding. But the information about the slot that executes the reference operation must be updated for each operation. This can be incorporated easily in the rebinding algorithm by keeping in mind the triple (i, tr, k) for each original operation op that was rebound in instruction i to slot k, where tr is the temporary register written by op. For a duplicated operation op' that was rebound to slot k' and writes into the temporary register tr', the corresponding triple is found by looking up for a triple with tr' in the second component. Then the bit field *refEU* in op' is updated to k. The operation op must be revisited such that the bit field *refEU* in op can be updated to k'. Due to this backward patching a complete basic block should be moved into the data memory before rebinding starts for the instructions of that basic block. Moreover, when rebinding operation k', then slot k must be excluded for executing operation k'.

This can be modeled in the rebinding algorithm by declaring the needed operator for executing op' as faulty in execution unit k'.

The fault state that is determined by the concurrent error detection can be used directly for the adaptation with the software-based rebinding, because faults are localized and handled at the same granularity level in both approaches. Thus, a diagnostic off-line test, as it is shown in figure 4-13, is not needed.

4.3.6.2 Modifications of the Software-Based Rescheduling Algorithm

When the software-based rescheduling is used in combination with the hybrid on-line approach, then also the bit field *refEU* of each original and duplicated operation must be updated. Furthermore, the data dependencies that arise from the temporary registers must be respected. This is incorporated into the rescheduling algorithm in the same way as for the regular registers with the lookup table *use* and *def*. Moreover, the order between an original operation and a duplicated operation is maintained, because both operations have the same destination register. This output dependency prevents the duplicated operation from being scheduled before the original operation. However, as a special case must be taken into account that both operations, original and duplicate, can be scheduled into the same instruction.

The software-based rescheduling can handle faults at read-port and at bypass level, but the hybrid on-line approach only at execution unit level. I.e., faults in the read ports and bypasses are detected by the hybrid approach with early rebinding, but they affect all operation types of a slot. I.e., when a fault in the read ports or bypasses of slot k occurs, then each operation executed in slot k may produce a wrong result, such that step by step all operators of that slot are declared as faulty, until $fsEU(k,t) = 0$ for each operation type $t \in \mathcal{O}$. Hence, such faults can be considered as a slot fault. However, if software-based rescheduling is used in off-line mode, then a finer grained software-based diagnosis can be performed before the software-based rescheduling is invoked (see figure 4-13). When the processor switches into the off-line mode, then the fault state determined by concurrent checking is discarded, and a finer grained diagnosis is performed that will localize, for example, the faulty read port. Then the fault state determined by the fine-grained diagnostic test is used for the software-based rescheduling. Hence, slot k is used again for executing operations. However, the on-line determined coarse-grained fault state is needed in the voting. Then the faulty slot must be excluded from using it for re-executing a failed operation.

4.3.7 Results

The hardware extensions shown in figure 4-7 were included in the VHDL code of the VARP processor with some limitations. For example, no counters were implemented for distinguishing permanent from transient faults. Once a fault is detected in an operator, then the operator is declared as faulty forever. Thus, the performance degradation during the execution of the user application is negligible, because at most two clock cycles delay occur for each operator that becomes faulty. The hardware overhead that is introduced in the VARP processor is not negligible. For this reason a reliability analysis is provided for various system configurations. In particular, five VARP processor systems are compared with each other:

- $VARP_{early}$: the VARP processor with early rebinding,
- $VARP_{late}$: the VARP processor with late rebinding,
- VARP: the non-fault tolerant VARP processor from figure 2-1,
- $VARP_2$: a non-fault tolerant VARP processor with two slots, and
- $VARP_{TMR}$: a TMR-system based on three $VARP_2$ processors.

The computational performance of the $VARP_2$ and the $VARP_{TMR}$ processors is almost equal to the computational performance of the $VARP_{early}$ and $VARP_{late}$ processors, because all of them can execute two original operations per clock cycle. Table 4-4 shows the cell area in μm^2 of the additional components needed for implementing the hybrid on-line approach. Please note that the size of the shadow-fetch-registers is subsumed in the rebinding logic of the $VARP_{early}$ processor.

Component	**$VARP_{late}$**	**$VARP_{early}$**
Rebinding Logic	1298	1547
4 x FDCL	550	550
FDCL Control	587	690
Temporary Register File	3879	3879
Total	**6314**	**6666**

Table 4-4: Cell area in μm^2 for the additional hardware components that were introduced in the VARP processor from figure 4-7.

For the reliability analysis regarding permanent faults, the components of the five VARP-systems are partitioned into two types; the non-fault tolerant and the fault tolerant components. If one of the non-fault tolerant components fails, the whole system will fail. Each fault-tolerant component belongs to a particular slot. The functionality of a slot with a permanently faulty component can be overtaken from another slot. C_{ft} denotes the total cell area of the fault-tolerant components of a single slot and C_{nft} denotes the cell area of the non-fault tolerant components. The

partitioning is shown in table 4-5. Thereby the cell area of a single execution unit is explicitly denoted by C_{EU}, because the reliability analysis of the five VARP-systems is done for varying sizes of the execution units. The used execution unit sizes are listed in table 4-6. Column n denotes the number of slots in each system. Hence, the total cell area of each system type is obtained by $C_{nft} + n \cdot C_{ft}$.

System Type	C_{nft}	C_{ft}	n
VARP	$24127 + 4 \cdot C_{EU}$	0	4
VARP_2	$13868 + 2 \cdot C_{EU}$	0	2
VARP_{late}	$24127 + 6314$	C_{EU}	4
VARP_{early}	$11199 + 6666$	$3232 + C_{EU}$	4
VARP_{TMR}	$7404 + 105$	$6464 + 2 \cdot C_{EU}$	3

Table 4-5: Cell area of the fault tolerant and non-fault tolerant components in the five VARP systems in µm².

The VARP and VARP_2 systems have no fault tolerant components. The VARP_{late} processor has only the execution units as fault tolerant components. It is assumed that all hardware extensions needed for implementing the hybrid on-line approach (listed in table 4-4) must be faultless for a functioning processor. For this reason their cell area is listed in column C_{nft} for the VARP_{late} and VARP_{early} processors. Due to the early rebinding, the VARP_{early} processor can handle faults in its bypasses, read ports and decode registers. The total cell area of these fault tolerant components is $3232 + C_{EU}$. The VARP_{TMR} system is obtained from the VARP_2 system by tripling the decode- and execution-stage of both slots of the VARP_2 processor. The cell area of the fault tolerant components in both slots is $6464 + 2 \cdot C_{EU}$. The voter that is needed at the end of the execution stage has a cell area of 105 µm².

The fault tolerant systems VARP_{late}, VARP_{early} and VARP_{TMR} will work properly as long as there is at most a single failing slot. Hence, all the five systems can be considered as a serial composition of the non-fault tolerant components with an (n-1)-of-n system that is composed of n slots (n is given in table 4-5). Furthermore it is assumed that $R_C(t) := e^{-\lambda \cdot t}$ is the reliability function for a single cell area. Then

$$R_0(t) := R_C(t)^{C_{nft}} \cdot R_C(t)^{n \cdot C_{ft}} = R_C(t)^{C_{nft} + n \cdot C_{ft}}$$

is the probability that all components of the system are faultless at time t. The probability that one of the slots is faulty is given by

$$R_1(t) := n \cdot R_C(t)^{C_{nft}} \cdot R_C(t)^{(n-1) \cdot C_{ft}} \cdot (1 - R_C(t)^{C_{ft}})$$

From both equations follows that the reliability function $R_S(t)$ for a VARP system S with n redundant slots is computed by

$$R_s(t) = R_0(t) + R_1(t)\,.$$

The reliability plots for the five systems are shown for three different execution unit types in figure 4-15 to figure 4-17. The plot in figure 4-15 shows the reliability functions, when small sized execution units are employed in the VARP processor. This is an execution unit that supports only 16-bit *add-/sub*-operations together with various logical operations.

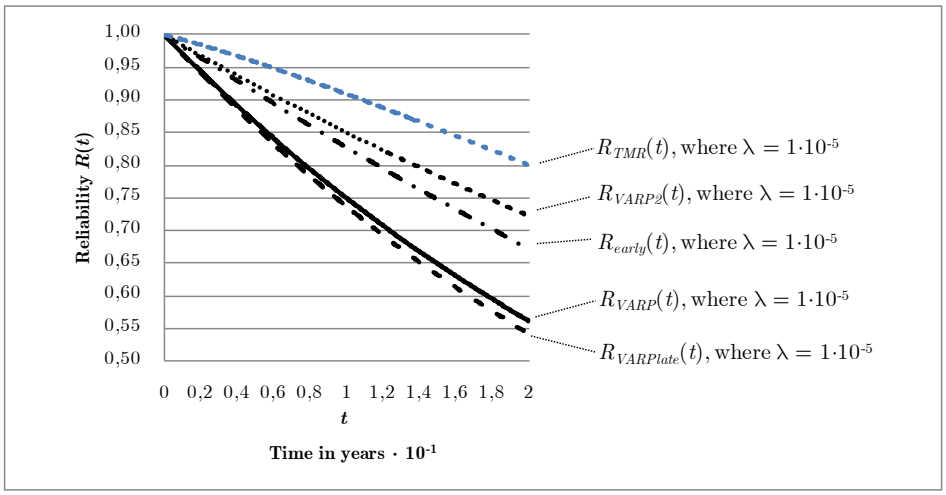


Figure 4-15: Reliability plots for the VARP systems with small sized execution units ($C_{EU} = 1171$).

The reliability of the VARP_{late} and VARP_{early} processors is inferior to the reliability of the non-fault tolerant VARP_2 processor that has the same performance. Moreover, the reliability of the VARP_{late} processor is even worse than the reliability of the non-fault tolerant VARP processor. This result follows from the size of the additional hardware components in the fault tolerant processors, which are too big compared with the size of the fault tolerant components. I.e., the proposed approach is not suitable for handling permanent faults in VARP processors with small sized execution units.

However, this situation changes for medium sized execution units that support a single cycle *mul*-operation. The plot of the reliability functions for these systems is shown in figure 4-17.

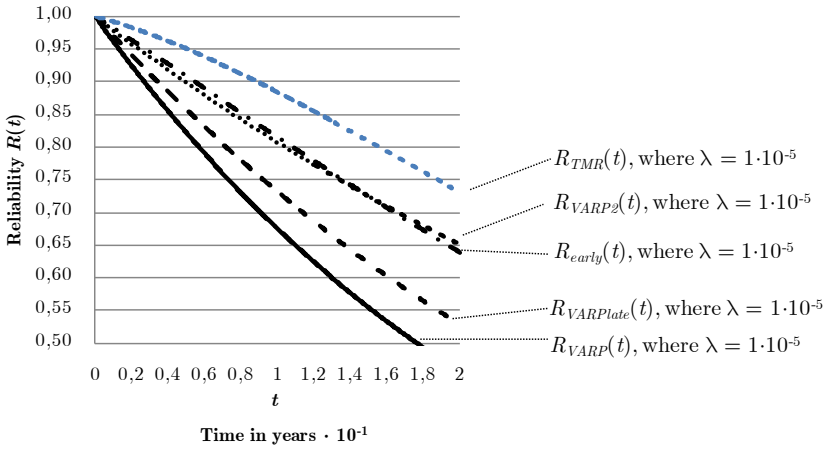


Figure 4-16: Reliability plots for the VARP systems with medium sized execution units ($C_{EU} = 3796$).

There the reliability of the VARP_{early} processor becomes slightly better than the reliability of the VARP_2 processor. Thus the proposed hybrid method increases the reliability of the VARP_{early} processor regarding failures due to permanent faults. However, the VARP_{early} system will not reach the reliability of the VARP_{TMR} system. Moreover, for medium sized execution units the size of the non-fault tolerant components in the VARP_{early} processor becomes smaller than the size of the non-fault tolerant components in the VARP_2 processor. I.e., the probability that a temporary fault causes a failure in the VARP_{early} processor is smaller than for the VARP_2 processor. Hence, the early rebinding also provides a hardening against temporary faults for medium sized execution units.

For larger scaled execution units[23] the reliability of the VARP_{early} system approaches the reliability of the VARP_{TMR} system, as it is shown in figure 4-17. Moreover, also the reliability of the VARP_{late} systems reaches the reliability of the VARP_2 system.

[23] Execution units that also support *div*- and *mod*-integer operations.

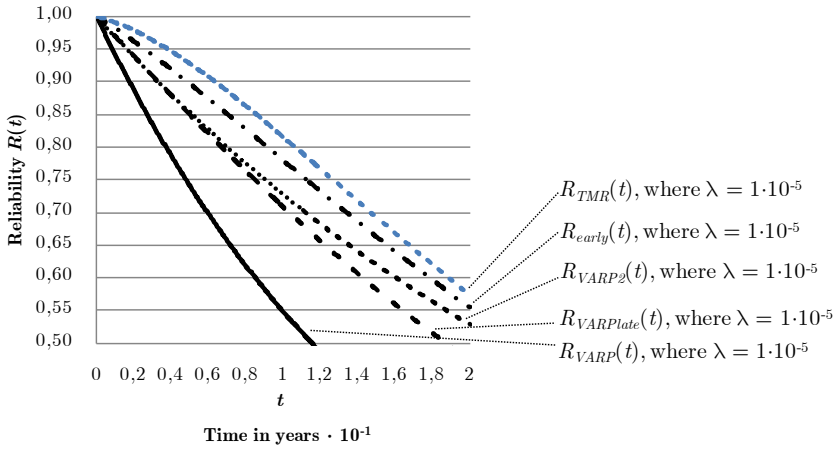


Figure 4-17: Reliability plots for the VARP systems with large sized execution units ($C_{EU} = 8987$).

It can be concluded that the VARP_{TMR} system provides in all three cases the best reliability, although for larger scaled execution units the VARP_{early} system approaches this reliability. But this benefit of the VARP_{TMR} system comes at the cost of an increased hardware overhead. From the cell area sizes in table 4-5 follows that the VARP_{early} processor has a fixed size of 30793 cells that grows with $4 \cdot C_{EU}$, and the VARP_{TMR} has a fixed size of 26901 that grows with $6 \cdot C_{EU}$. Table 4-6 shows in column *overhead* for various execution unit sizes C_{EU} the hardware overhead of the VARP_{TMR} processor compared with the VARP_{early} processor.

C_{EU}	**Size VARP_{early}**	**Size VARP_{TMR}**	**Overhead**
1171	35477	33927	-4 %
3796	45977	49677	8 %
8987	66741	80823	21 %

Table 4-6: Hardware overhead comparison of the VARP_{early} and VARP_{TMR} systems for varying execution unit size.

For small sized execution units the VARP_{TMR} is smaller than the VARP_{early} processor. Therefore, the overhead is negative. But for medium and large scaled execution units the overhead becomes 8% and 21% respectively. This overhead will further increase with the size of the execution units. For example, supporting floating point operations increases further the size of the execution units, which will further increases the attractiveness of the VARP_{early} processor. Moreover, the proposed VARP_{early} processor provides more flexibility. For example, the compiler can trade performance against reliability, because not every operation must be duplicated. By this some code sections of the user application can be protected against temporary faults, while some other portions are not protected, but they will use the available redundancy in the data path of the processor for higher performance.

4.3.8 Conclusions

The hybrid on-line approach provides on-line fault detection and error recovery capability for the VARP processor. This is achieved by concurrent error detection such that a transient or a permanent fault is detected on-line. The used idea is conceptionally very similar to the one published in [24], where operations are duplicated in software, but comparison of the results is done with hardware support. However, the work in [24] is substantially extended by the proposed hybrid approach, because also a fast error recovery is provided in hardware. Error recovery is possible within one to two clock cycles, depending on the position of the rebinding logic. On-line detected permanent faults can be masked in order to avoid fault recovery the next time. Thus the presented approach can be used for the design of reliable real-time systems, because the delay due to the recovery procedure is very small. Moreover, the architecture provides the opportunity that the compiler or assembler programmer can trade performance against reliability by selecting only certain operations for duplication. Thus some parts of an application may run with high reliability by using redundant resources for redundant computations, while other parts benefit from higher performance by using redundant resources for higher instruction level parallelism.

The hybrid on-line approach has also some limitations. First, after the detection of a permanent fault in an operator, the fault detection capability of the system gets partly lost. I.e., it is no longer possible to check the result of these operations that are a reference operation of an operation that is executed on the faulty operator. Please note that further faults in all other operators that do not use the faulty operator as reference operator can be detected. However, it was shown that the partly loss of the fault detection capability is overcome by additionally performing off-line a software-based self-repair, such that all faulty operators are not used anymore. By this the system is in degraded fault detection modus after the detection of a permanent fault. But the full fault detection capability is restored after the adaptation of the user application to the current fault state, which can be done either during execution breaks or during the next startup phase.

A second limitation of the presented approach is the amount of administrative hardware overhead needed for on-line fault handling. Because these administrative components are not redundantly available, they have a negative impact on the reliability of the processor. For the VARP processor this has the consequence that some portions of the data path can be protected against temporary faults. But the administrative hardware may be affected by permanent faults after some time, which increases the probability that the processor fails due to a permanent fault in the administrative hardware. This effect is only avoided, when the size of the fault

tolerant components becomes big enough. For the considered VARP processors this is achieved with medium to large scaled execution units.

4.4 Summary

In this chapter the application of the software-based self-repair in cross-layer fault tolerance approaches was demonstrated by supporting hardware-based fault tolerance techniques. Both methods fit well together, because hardware-based methods are active in on-line mode, and software-based methods are active in off-line mode. By this, the self-repair functionality at software-level assists the techniques used at hardware-level. In particular, the combination of hardware- and software-based administration of redundancy allows for a better utilization of the remaining functional components of the processor, because the software-based methods perform a finer-grained reconfiguration than the hardware-based methods. For example, hardware-based rebinding that suffers from performance degradation, when the program is adapted dynamically to the current fault state, can be improved in this way. Also the concurrent error detection capability of the hybrid on-line approach can be restored by the application of software-based methods. Due to the support by software-based methods, a coarse-grained fault handling by hardware-based methods is sufficient. The software-based method allows for a finer grained reconfiguration, not immediately after concurrent error detection, but at carefully specified points in time; for example during the startup of the processor or during execution breaks. At these points in time, the software-based methods have enough time for a more detailed fault diagnosis and reconfiguration of the system. This time is either not available during on-line error handling or performing such a detailed fault diagnosis and reconfiguration on-line would require much more hardware resources. Hence, administrative hardware overhead is reduced by shifting complex reconfiguration tasks in software.

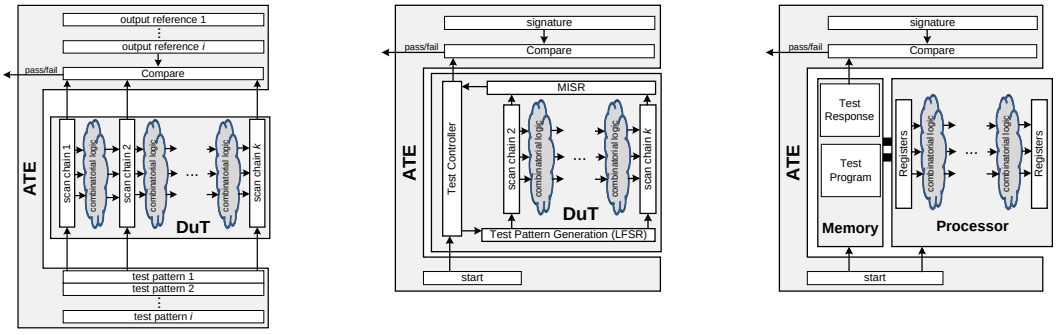
Chapter 5

Adaptive Diagnostic Software-Based Self-Test

So far the fault state of the VARP processor was formally modeled by the fault state functions. These fault state functions, which are represented by fault state registers in the VARP processor, are needed by the self-repair routines for adapting the user application to the current fault state of the processor. In the previous chapter an on-line method for determining these fault state functions at coarse-grained level was presented. However, a finer grained method is needed that can also work off-line for providing the information needed by the fine-grained self-repair approach presented in section 3.3. I.e., the self-test must be able to localize permanent faults in multiple components with the same granularity that is used for fault handling by the software-based rescheduling. For this reason a survey of existing test methods that can be used for diagnostic purposes is given. Moreover, the limitations of diagnostic self-test methods are described, when they are used in a system that employs software-based self-repair techniques. This will be a motivation for the novel *adaptive* part of the software-based self-test. Based on these considerations, a system configuration is described that allows for detecting and localizing multiple permanent faults during the startup of the VARP processor by executing a software-based self-test routine. The achieved fault coverage and diagnostic capability of the self-test method are presented. By putting together the adaptive diagnostic self-test and a software-based self-repair method, a comprehensive software-based self-test and self-repair method for a statically scheduled processor is obtained.

5.1 Related Work

In order to determine the fault state of the VARP processor in the field, the correct functioning of particular components of the processor must be tested by a method that is simple enough to be executed by the system itself without adding too much hardware. Moreover, the test must provide diagnostic results at that granularity level that is used by the self-repair routine. The diagnostic test is executed in off-line mode during the startup phase of the system or during execution breaks. I.e., the processor does not execute the user application at that time, such that most of the processor resources can be used exclusively by the test procedure. Such a scenario appears during the manufacturing test, too. Manufacturing test in volume production is usually a simple pass/fail test for detecting faulty devices. The goal of a diagnostic test is the localization of the fault. The diagnostic information can be provided at structural or functional level. *Structural diagnostic information* localizes the fault site with respect to a given structural fault model, for example at gate level. *Functional diagnostic information* determines whether or not a particular functionality of a component is provided correctly. Existing diagnostic off-line techniques are strongly related with structural and functional test methods used for manufacturing test. Their advantages and disadvantages for diagnostic test in the field are discussed in the subsequent sections.



(a) (b) (c)

Figure 5-1: (a) Scan test. (c) Scan-based built-in self-test (BIST). (d) Software-based self-test (SBST).

5.1.1 Structural Test and Diagnosis

Diagnostic manufacturing test is under investigation since the 1960's [36] in order to localize manufacturing flaws. It relies on the same test infrastructure that is used for manufacturing test without diagnosis. This is some kind of tester (ATE) that wraps the *device under test* (DuT) as it is shown in figure 5-1. *Test patterns* are consecutively applied to primary inputs of the DuT. The corresponding test response is captured at the primary outputs and compared with reference values.

Mismatches indicate a fault. Forcing a mismatch for particular faults becomes difficult for larger sequential circuits. For this reason various design for testability methods were proposed for improving the controllability and observability of internal states of the DuT [142]. Among these methods are scan-test designs [52, 65], shown in figure 5-1 (a), probably the most popular ones. Scan test is a *structural test* method, because the DuT is not operated in normal functional mode for the test. During the test mode, the memory elements of the DuT are connected to one or more *scan chain*(*s*). For a full scan design these scan chains divide the DuT into combinatorial logic blocks that receive their inputs from the scan chains, and write their outputs into the scan chains. Scan chains are used for applying test patterns to the combinatorial blocks by shifting in test patterns, and they are used for capturing test responses by shifting them out after a functional cycle. Usually the test patterns applied during the scan test are *structural test patterns*. These are test patterns that are generated by using structural information about the combinatorial logic blocks in the DuT. Based on techniques developed during the past decades [70, 150], the process of test pattern generation is nowadays automated by *automated test pattern generation tools* (*ATPG-tools*). By knowing the internal structure of a DuT with respect to a given structural fault model, the quality of a *test set*, i.e., a set of test patterns, can be expressed by the *fault coverage*. On the other hand, structural test patterns may provide inputs that will never appear in functional mode, which causes an overtesting.

The concept of a scan test can be used potentially in the field by employing a *structural built-in self-test* (*BIST*) [93]. The BIST was developed for overcoming the limitations of the slow interface between the ATE and the DuT during manufacturing test. Figure 5-1 (b) shows the STUMPS architecture for a BIST [17]. Test patterns for the scan chains are generated on-chip, for example, pseudo randomly with a *linear feedback shift register* (*LFSR*) or with arithmetic units [142]. In order to improve fault coverage and to reduce test time, additional deterministically generated test patterns may be stored in an on-chip memory [86]. Test responses are compacted on-chip into a *signature*, e.g., with a *multiple input shift register* (*MISR*s). The ATE only initiates the BIST and retrieves the signature(s). Hence, most of the functionality of the ATE from figure 5-1 (a) is moved on chip, such that the BIST infrastructure can be used in principle for performing a self test autonomously in the field.

Both scan test and scan-based BIST infrastructure are used for diagnosis with emphasis on post production diagnosis for detecting systematic flaws in the manufacturing process. Thereby the only available diagnostic information comes from the test responses. Diagnosis is the mapping of test responses to faults. The best diagnostic resolution is achieved, if each fault will produce a different test response with respect to a given fault model. Particular methods have been

developed in order to improve the diagnostic resolution of a set of test patterns [31, 73]. Fault dictionaries provide a simple and fast way for mapping test responses to faults [147] during production test. They are created by fault simulation, most likely for single fault models. Therefore the method is very sensitive to unmodeled faults that produce test responses that cannot be found in the fault dictionary. In this case the diagnosis will fail. Moreover, fault dictionaries, and even more compact representations like fault trees [26, 27], become very large. For example, the size of the full fault dictionary for a relative small ISCAS89 benchmark circuit s35932 is 9532 MB [25], and the size of a fault tree that takes only pass/fail information of each test pattern into account is bounded by $F \cdot (T + 1)$, where F is the number of collapsed faults in the DuT and T is the number of test patterns [38]. The size of fault dictionaries and fault trees does not matter for post production test, when diagnosis is done off-chip, but it becomes crucial for on-chip diagnosis, when information must be stored on-chip.

This situation does not change for diagnostic scan-based BIST methods, where test responses are available on-chip as signatures. The problem of diagnosis becomes even more complex for diagnostic scan-based BIST approaches, because test responses for all test patterns are typically compacted into a single signature, and this single signature does not provide enough diagnostic information for fault localization [45]. Various fault isolation techniques were proposed for overcoming this problem. Faults are isolated by performing additional test sessions that use different test pattern sets [188] or compute a signature only for specific scan chain elements [178]. Based on these signatures, fault candidates are determined either directly by mapping faulty signatures to fault candidates [42] or indirectly, by determining the faulty scan chain contents first [105]. From these scan-chain contents fault candidates in the combinatorial logic are computed [6]. However, in any case, this mapping is not done on-chip. The runtime of the algorithms used for such a diagnosis ranges between 60 and 125 seconds, measured on workstations [42]. Obviously this problem cannot be solved in the field from a simple embedded system. For this reason, diagnostic scan-based BIST methods only collect information on-chip that is used for off-chip diagnosis [45, 54]. The general problem of using diagnostic BIST approaches in the field is the fine-grained structural diagnosis at gate-level. This causes two inherent problems. First, the gate level granularity increases the required memory consumption for diagnosis and the runtime for fault localization. Therefore, diagnostic BIST approaches rely on powerful off-chip analysis techniques, such that diagnosis cannot be performed on-chip. Second, the structure-oriented test method works on gate level net lists that do not consider processor components at functional level. Mapping a fault localized at gate level on processor components is not in the scope of these works. Hence, the link from a localized fault in the structural fault model to the caused

functional misbehavior of the component containing that fault is not provided by structural testing. But this information is needed for the self-repair algorithms.

For BIST approaches that perform diagnosis at a coarser grained level, e.g. at component level, the complexity of diagnosis is reduced. In [169] test patterns for components of pipeline stages are stored in an on-chip ROM. During idle cycles, these test patterns are applied as input to the components, and the test responses are compared with reference values. Vierhaus et al. proposed in [95] a scan-based BIST technique for VLIW processors that employs the redundancy in the data path. Test patterns are generated with LFSR structures, but test responses are not compared with pre-computed test responses. Rather, the same test pattern is applied to all slots of a VLIW processor, and the obtained results are compared with each other. A similar idea is used in [183]. There, the test patterns are stored in the program memory as if they were instructions. Due to the used VLIW processor architecture, these test patterns can be fetched directly into the pipeline for testing the pipeline, and the data path of the VLIW processor as if they were normal instructions. Test responses are checked at the end of the pipeline by comparing them with each other in a similar way as it was proposed in [95]. A special test mode prevents the processor from treating some of these test patterns as branch or illegal operations. The advantage of these BIST approaches is that they provide diagnostic information at coarser grained level, e.g., a particular slot of the VLIW processor is faultless or not. But, performing a majority vote on the test responses fails for multiple faults in the slots of a VLIW processor. Moreover, these approaches are too coarse-grained. I.e., they will not achieve the diagnostic resolution needed for the fine-grained self-repair at reasonable costs.

5.1.2 Functional Test and Diagnosis

Software-based self-test (*SBST*) is a *functional BIST* method. It was first proposed by Abraham and Thatte [179]. In contrast to structural BIST approaches, SBST is non-intrusive, i.e., no additional hardware is needed in the DuT. Moreover, the DuT must be a processor, and the self-test is a program that is executed on the processor. The processor is operated in functional mode at speed. Functional testing has various advantages. For example, overtesting is avoided, because only functionally legal inputs are provided to the DuT, and it is well suited for detecting unmodeled faults. As shown in figure 5-1 (c), the test program must be loaded into the program memory, which can be accomplished during manufacturing test, for example, by the ATE. Executing this test program can be considered as a way of originating *functional test patterns* in the internal registers of the processor with that program. Test responses in the internal registers are captured by an instruction sequence that computes a signature from these results, and finally stores this signature in the data memory, such that the ATE can check

this signature. The most difficult problem when using a software-based self-test is the generation of high quality test programs with reasonable efforts. According to [141] various methods exist for the generation of test programs. Test programs may be generated manually [185] or randomly [167] without the knowledge of the internal structure of the processor. Then no measure for the quality of the test is available. In order to reduce the size of the test programs that must be stored or transmitted to the memory of the processor under test, the test program can be generated on chip. The FRITS approach in [132] uses a random based method, where a small kernel is executed by the processor under test. This kernel generates pseudo-random test programs. By this, only the small kernel must be loaded in the program memory, which reduces the test time.

With knowledge about the internal structure of the processor under test, the quality of manually or randomly generated test programs can be quantified by fault simulation. I.e., for each possible fault the test program is simulated with the processor model containing this fault. In this way the number of faults that can be detected with a set of test programs can be determined, and, hence, the fault coverage of these test programs. With these results the randomly generated test program set can be improved in an iterative way [46]. Feedback information from fault simulation is used in such approaches for reducing the test program size, while improving the fault coverage.

Knowing the internal structure of the processor allows also for deterministic test pattern generation with ATPG tools. This can be used for deterministic test program generation [181]. Templates of instruction sequences are used, where particular bit-fields of the instructions are left blank, for example for operand values. These bit fields must be filled with appropriate values, which are generated with ATPG tool support. Constraints are used for specifying restrictions imposed by the template for test pattern generation. By this, the process of test pattern generation for complex logic functions is shifted into an ATPG tool, instead of being done manually or randomly. In [37, 39] this idea is extended such that the process of template selection and test program generation can be automated. Moreover, random and deterministic test program generation may be integrated into hybrid approaches [97] in order to improve the fault coverage.

Basically, a functional test method like SBST is well suited for the diagnostic functional test of a particular component of the processor in the field. Test programs for various processor components were published recently [98] and achieve high fault coverage. In [69] a diagnostic method for the read ports of a register file is presented. A software-based self-test for the register file of a VLIW processor is presented in [151]. Test programs for the software-based test of the translation look-aside buffer and branch prediction unit were published in [152, 180]. By putting together all of these test programs, high fault coverage for

complex processors may be achieved. Unfortunately, a good diagnostic test program is not obtained by putting together all these test programs. The diagnostic capability of such a test program is limited, because even a simple test program uses many processor components, such that a faulty component is not uniquely identified. For example, consider the simple test program from figure 5-2 for the adder in slot 3 of the VARP processor. The test program is composed of some initialization code, test code, and capturing code. The *initialization code* loads some values into registers *r0* and *r1*. The *test code* will utilize the adder, and the *capturing code* makes the result observable. The problem of that piece of test code is that it will not only check the adder, but it will also check many other components of the processor. For example, the initialization code uses some components of slot 1 and the register file. If there is a fault in a pipeline register of slot 0, then the initialization is not executed correctly, but the test program will detect a fault in the adder of slot 3.

Figure 5-2: Example of a piece of test code for the adder in slot 4.

Thus, a faulty response produced by a test program will not indicate necessarily a fault in the tested component. More sophisticated diagnostic techniques are needed. Chen proposes in [38] that for good diagnosis a complete test routine is composed of many test programs. Each test programs utilizes only few components of the processor, such that a failing test program refers to a small set of faults only. On the other hand, all the test programs together should cover as many faults as possible. In order to achieve both goals, Chen describes in [38] a method for selecting test programs from a given set of test programs. The test programs were generated manually. From the pass/fail information of each test program, the set of fault candidates is isolated by using fault trees. A major problem of this approach is the generation of small diagnostic test programs. Bernardi et al. presented in [21, 22] a method for automatically improving diagnostic test programs. The original test programs were derived from ATPG generated test sets. During an iterative phase the diagnostic resolution of the test programs is improved with genetic algorithms. In both approaches the execution order of the test programs does not matter, because diagnosis is based on fault trees, where

only pass/fail information of each test program is considered. The quality of the diagnostic resolution of both approaches is determined at gate level by fault simulations. Therefore, both approaches will provide structural diagnostic information at gate level. Mapping this structural information to functional diagnostic information is not in the scope of these works, because both methods are dedicated to manufacturing test and diagnosis. The link between the detected faults at gate level and the functional misbehavior caused by these faults is missing. However, this link is very important for a diagnostic in field self-test that must provide fault state information for a subsequent self-repair method.

5.2 Towards a Systematic Adaptive SBST Routine

As described in the previous section, structural BIST approaches suffer from a computational expensive diagnosis that cannot be performed on-chip. Moreover, the diagnostic information is too fine grained. Functional BIST approaches, i.e., SBST, allow for testing at the required granularity level, but test programs suffer from the usage of too many processor components, which makes the unique diagnosis of a fault difficult. Finer-grained diagnostic SBST approaches produce diagnostic information at structure-oriented level. This makes the result of such diagnostic approaches useless for self-repair approaches that need functional diagnostic information.

For this reason, the starting point for developing a diagnostic self-test for the VARP processor is the self-repair method. This self-repair method determines the granularity for the test of the components and the tested functionalities of each component. Moreover, these components must be tested in a systematic way. In the best case, the test of a particular component only relies on the usage of already tested components, except the component under test. For example, consider again the example in figure 5-2. In order to make sure that only the adder is tested, all the other processor components used by the test program should be tested before. If they are faultless, then a detected fault can be allocated to the adder.

Another problem of existing diagnostic SBST approaches is that the used test programs are generated statically. This becomes a problem, because faults in multiple processor components can be handled by the presented software-based self-repair method. Thus, the diagnostic SBST must be able to detect and localize faults accurately, even in the presence of already detected faults. But if an already detected fault affects, for example, the initialization code of a test program, then the component under test cannot be tested correctly. As an example consider again the test program in figure 5-2. This test program will not produce accurate diagnostic results for the adder, if the initialization of register *r0* and *r1* is done in

a wrong way due to a fault in slot 0. The solution to this problem is the adaptation of the test code. The adapted test program is shown in figure 5-3.

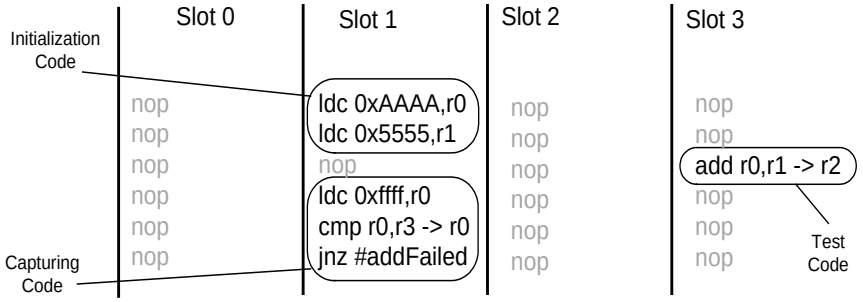


Figure 5-3: Test code from figure 5-2 adapted to a fault in slot 1.

All the initialization and capturing code for the test was moved into slot 1, where the code is executed correctly. Only the test instruction remains at their original position. Please note that moving the test operation to slot 1, too, would corrupt the test program, because the *add*-operation is then executed in slot 1, and therefore the adder of slot 1 is tested. But the test routine would still be used for testing adder 3. It follows that the adaptation of the test routine should be done in such a way that the initialization and capturing code is executed on faultless components of the processor, and all test instructions remain at their original position. Therefore, the self-test routine for the VARP processor must be *adaptive* in that sense that it is adapted to the current fault state of the processor. Thereby, a diagnostic test can be performed by the adapted test routine, even if the original test routine would deliver wrong diagnostic results. Putting together the systematic execution of small test programs and the adaptation of these test programs yields a general test flow for the startup phase that is illustrated in figure 5-4.

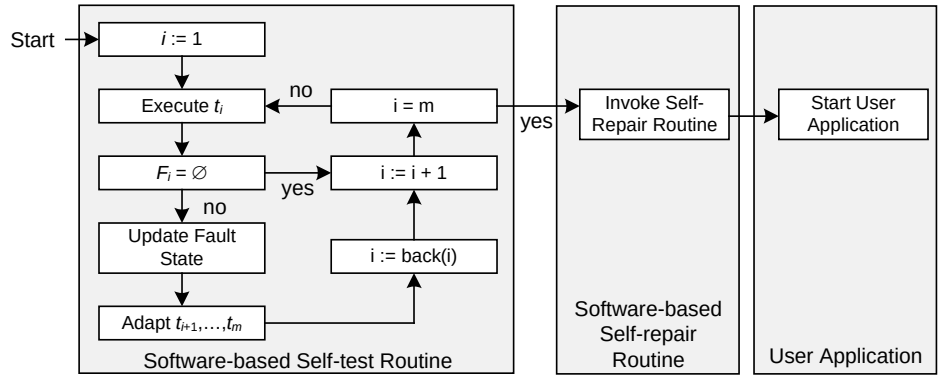


Figure 5-4: General test flow of a systematic adaptive software-based self-test routine.

Suppose the self-repair routine can handle faults in processor components $c_1,\ldots,c_m$. Then for each component c_i a test program t_i must be provided. Test program t_i checks the functionalities of c_i. Component c_i may provide various sub-functionalities that can be tested with t_i. For example an execution unit supports

various operations. Then each provided operation can be considered as a sub-function. Let F_i denote the sub-functions provided by c_i that were detected to be faulty by t_i. Each test program t_i is separated in *initialization code*, *test code*, and *capturing code*, as it is shown in figure 5-3. The test programs are executed in ascending order $t_1,\ldots,t_m$. In the ideal case t_i uses only components c_k, $k < i$, for initialization and capturing that were tested before by $t_1,\ldots,t_{i-1}$. Moreover, when t_i detects a fault, then test programs $t_{i+1},\ldots,t_m$ are adapted in such a way that they do not use the faulty sub-functions F_i of component c_i. In practice, it will be not possible to find an execution order, where each test program uses only tested components for initialization and capturing. Then it is accepted that there are test programs t_i that use untested components c_k. When such a component c_k is detected later as faulty, then, as a general solution, the test routine can be set back to the test of component c_i with the knowledge that c_k is faulty. This is depicted in figure 5-4 by the node $i \coloneqq back(i)$. In section 5.4.3 it will be shown that there are also other solutions based on observed signatures for solving this problem. When the self-test routine is finished, the actual fault state of the processor is known and can be used from the software-based self-repair routine for adapting the user application.

5.3 System Architecture

The proposed test flow from figure 5-4 requires an administrative instance that invokes the test programs t_i and performs, if necessary, an adaptation of the remaining test programs. The administration and adaptation must be done in a faultless manner, but the components of the processor under test are tested incrementally, such that, especially at the beginning of the test routine, for many components the fault state is unknown. For this reason the administrative functionality is provided by a separate service core. The system architecture used for executing the systematic adaptive SBST is presented in figure 5-5.

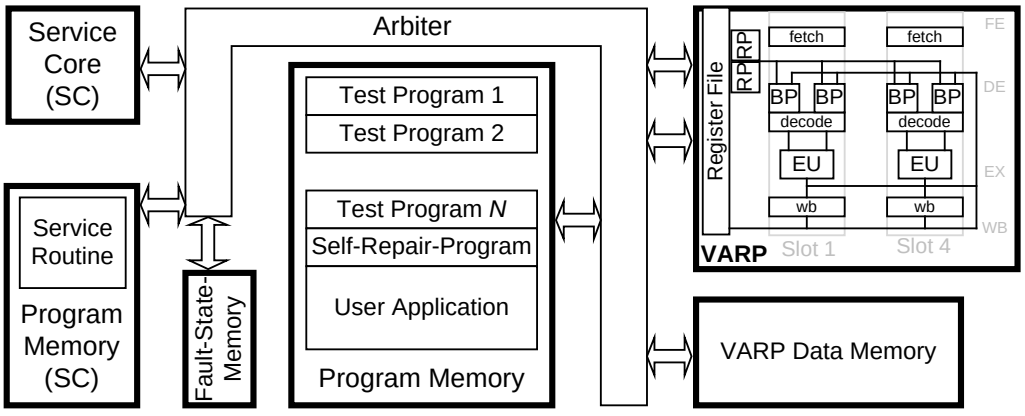


Figure 5-5: System architecture for the diagnostic adaptive software-based self-test of the VARP processor.

The system with the VARP processor is extended by the following components:

- The program memory of the VARP processor contains the diagnostic test programs t_i and a software-based self-repair program.
- The *fault-state memory* contains up to 64 64-bit *fault state registers* fsr_i, each of them storing the fault state of a particular component of the VARP processor. The bits of these registers correspond to the fault states modeled by the fault state functions *fsSlot*, *fsEU*, *fsRP*, *fsBPex*, *fsBPwb*, and *fsRF*. The bits of these registers are set by the test programs, when they are executed from the VLIW core. For this purpose, the VLIW core has been extended with special *check-instructions* that provide access to the fault state memory.
- The *service core* (*SC*) is a very small 8-bit processor. It is used for administrating the self-test of the VARP core. For this purpose, the SC executes the service routine, which is stored in the program memory of the SC. Furthermore the service core has read- and write-access to the program memory of the VARP processor, such that it is able to modify the self-test programs, which are stored in the program memory of the VARP processor. The SC has also access to the fault-state-memory, such that it has access to the current fault state of the VARP processor for adapting the self-test programs.
- The *arbiter* is responsible for organizing the memory accesses of the VARP processor and the service core. In particular it moves instructions from the VARP program memory into the SC-memory or into the VARP data memory. These instructions can be modified there. Then they are written back into the program memory.

Please note that it is the goal to determine the fault state of the VARP processor. For this reason a diagnostic test is only needed for those components of the VARP processor for which a self-repair method exists. As a consequence, a test of the control logic of the VARP processor is not in the scope of the presented system architecture. In particular such a test is not needed, if the control logic is made fault tolerant by a passive hardware redundancy scheme, as it was discussed at the end of section 3.4. Moreover, a diagnostic test of the test infrastructure shown in figure 5-5 is also not in the scope of this chapter, because the presented adaptive diagnostic self-test will be only useful for the VARP processor in the system. In order to perform a test of the test infrastructure, any other method may be better used. For example, some kind of scan-based BIST can be employed for testing the test infrastructure, because a simple pass/fail information is sufficient. If the test infrastructure passes the BIST, then the diagnostic adaptive self-test is started. Otherwise the system has a failure. Alternatively, a software-based self-test of the

test infrastructure may be performed instead of a scan-based BIST. For this reason the service core can execute a software-based self-test routine that just performs a simple pass/fail test of the service core. However, a test of the test infrastructure is not further investigated. Rather the facilities of an adaptive diagnostic self-test of the VARP processor will be investigated.

In order to perform an adaptive diagnostic self-test of the VARP processor an implementation of the general test flow from figure 5-4 was presented in [164]. In that work, the pipeline of the VARP processor was extended by so called 0/1-checkers that were used for detecting stuck-at faults in the pipeline registers. In [153] it was shown that the self-test can be executed without this hardware extension of the VARP processor. Hence, the only hardware extension of the VARP processor is an instruction set extension, by which the VARP processor has access to the fault state memory.

5.3.1 Instruction Set Extension

The instruction set of the VARP-processor is extended with the four *check*-operations shown in table 5-1 for supporting the software-based self-test. These operations are used for testing the data transport in the data path of the VARP processor and for accessing the fault state memory.

Instruction	Meaning
chkL0 rx,fsr_n,m	If executed in slot k, the value of register x is provided as left operand for EU k, and bit m of fsr_n is set to 1, if this register value is distinct from 0.
chkL1 rx,fsr_n,m	If executed in slot k, the value of register x is provided as left operand for EU k, and bit m of fsr_n is set to 1, if this register value is distinct from -1.
chkR0 rx,fsr_n,m	If executed in slot k, the value of register x is provided as right operand for EU k, and bit m of fsr_n is set to 1, if this register value is distinct from 0.
chkR1 rx,fsr_n,m	If executed in slot k, the value of register x is provided as right operand for EU k, and bit m of fsr_n is set to 1, if this register value is distinct from -1.

Table 5-1: New SBST related VARP-instructions.

Please note that the value -1 in table 5-1 means a bit vector, where all bits are 1. For a 16-bit data path this is a bit vector with 16 1-bits. The *check*-operations can be executed by all execution units of the VARP processor in parallel. They are encoded and executed in the same way as all other arithmetical and logical operations. I.e., the value of register x may be provided through the bypass or through the read ports from the register file, such that these *check*-instructions are very useful for testing the read ports and bypasses of the VARP processor.

5.3.2 The Service Core

The adaptive SBST is controlled by a small service core. The service core executes a service routine that is responsible for activating the execution of particular test programs. Each of these test programs t_i that are executed by the VARP processor will perform the diagnostic test of a particular component c_i of the VARP processor. If a fault is detected in such a component, then the service core is responsible for adapting the remaining test programs in such a way that their initialization and capturing code is executed on faultless components of the VARP processor. For this reason, the service core has access to the program memory of the VARP core. This access is granted by the arbiter. The communication between both cores is organized by two signals: *halt* and *continue*. The *halt*-signal is set to 1 by the VARP core, when it executes a *halt* instruction that freezes the program execution of the VARP core until the *continue*-signal is set to 1 by the service core. The communication of both processors during the startup phase is shown in figure 5-6, including the test programs executed by the VARP processor.

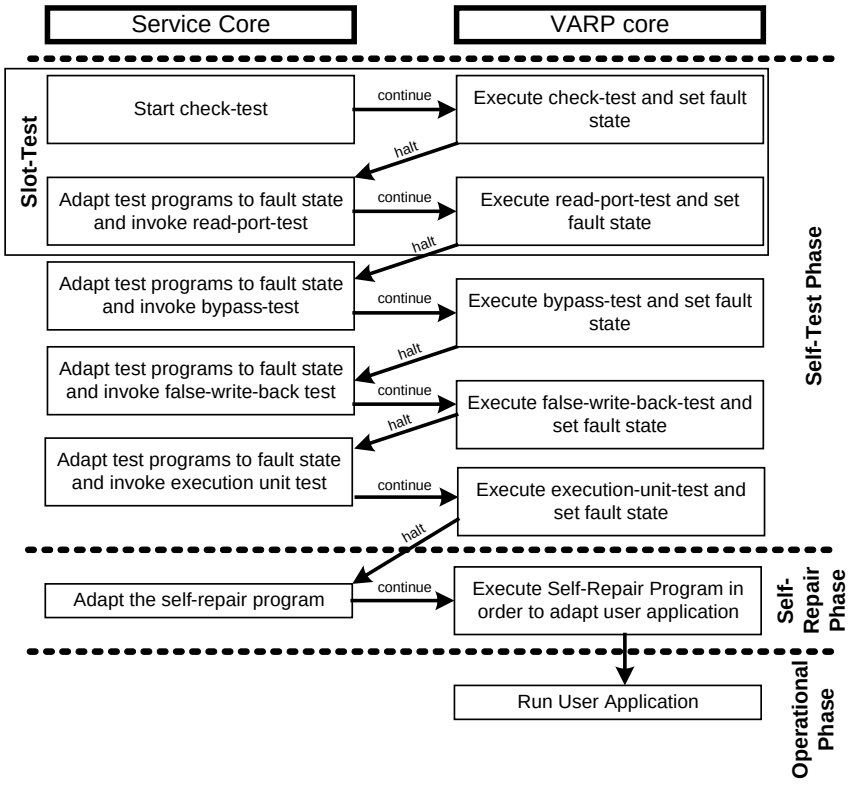


Figure 5-6: Implementation of the adaptive software-based self-test flow for the VARP processor.

These test programs are responsible for the following diagnostic tests:

- The *check*-test checks for the correct execution of the *check*-operations, because the correct result of the self-test strongly depends on the correct execution of them. This test will detect faults in some parts of the pipeline registers.

- The *read port test* checks the sub-functions of each read port. Each read port provides 64 sub-functions, whereby sub-function i corresponds to reading the value of register i through the read port. For this reason, the read port will also detect faults in the registers of the register file. Furthermore, the read port test will detect some faults in the pipeline registers that were not covered by the check-test.
- The *bypass test* checks two sub-functions of each bypass: The first sub-function is the forwarding from the execution or write-back stage into the decode stage. The second sub-function is the non-forwarding; i.e., the bypass passes the value from the read port.
- The *false-write-back-test* checks for each register of the register file that the register is not accidently written with a value, although this register should not be loaded with a value.
- The *execution unit test* checks the provided operations of each execution unit.

First, the service core activates the *check*-test for the VARP processor and waits until the *check*-test is finished. This is the case when the *halt*-signal is set to 1 by the VARP processor. Now the service core evaluates the fault state. When the *check*-test has detected some faults, then the service core adapts the remaining four test programs. After that the *read port* test is activated. Both read port test and *check*-test together constitute the *slot test*, because some faults in the pipeline registers are detected by the *check*-test and some other faults in the pipeline registers are detected by the read port test. Most faults in the pipeline registers are handled by the self-repair routine by avoiding the usage of that slot. After passing the slot-test, the bypasses and registers of the register file are tested. After that the data transport through the data path and the register file of the VARP processor has been tested, such that finally the operations of the execution units are checked. If correct test data can be provided to the inputs of the execution units, then testing them becomes straightforward. The obtained fault state is also used for adapting the self-repair routine. Finally, the self-repair routine is executed on the VARP core and adapts the user application. Please note that it would be also possible to execute the self-repair routine on the service core. However, the instruction set of the service core has been optimized for the relative simple adaptations of the test programs and the self-repair routine. Their adaptation is simple, because these programs have a very regular form such that for adaptation only single operations must be moved from one slot to another slot within the same instruction. For example, the self-repair routine is a totally sequential program that contains only one operation per instruction.

5.4 Test Programs

The subsequent sections describe the details of the test programs that were sketched on the right side of figure 5-6. The test programs were developed manually using the stuck-at fault model.

5.4.1 Check-Test

The *check-test* will check various parts of the pipeline registers of the VARP processor. The objective is to make sure that any fetched opcode, including the source and destination register number, will reach the execution units unchanged. This is of particular importance for loading immediate values into register with a *ldc*-operation, because the immediate value is encoded in the bit-fields for the source-register numbers of an operation (see figure 2-2). Figure 5-7 (a) shows the assembler code of the test program for slot 0, and figure 5-7 (b) shows the binary encoding of the assembler program. This test program is executed after a reset of the VARP processor. Therefore, all the registers *r0* to *r63* should have the value 0.

slot 0	slot 1	slot 2	slot 3
chkL1 R0, fsr_7, 0;	nop;	nop;	nop
chkL1 R0, fsr_8, 0;	nop;	nop;	nop
chkR1 R0, fsr_7, 63;	nop;	nop;	nop
chkR1 R0, fsr_8, 63;	nop;	nop;	nop
halt;	nop;	nop;	nop

(a)

slot 0				slot 1	slot 2	slot 3
opc	*src1*	*src2*	*dst*			
00001010	000000	000111	000000	00 … 00	00 … 00	00 … 00
00001010	000000	001000	000000	00 … 00	00 … 00	00 … 00
00001001	000111	000000	111111	00 … 00	00 … 00	00 … 00
00001001	001000	000000	111111	00 … 00	00 … 00	00 … 00
00000111	000000	000000	000000	00 … 00	00 … 00	00 … 00

(b)

Figure 5-7: (a) Assembler code for the check-test of slot 0. (b) Binary code for the check-test of slot 0.

The code sequence executes various *chkL1*- and *chkR1*-operations in slot 0. These operations read the value of *r0* and compare this value with -1 (see table 5-1). Obviously the value of *r0* should be distinct from -1. For this reason, bits 0 and 63 of fault state register 7 (fsr_7) and fault state register 8 (fsr_8) are set to 1, when this test sequence is executed correctly. These four bits are checked by the service core. When they are set to one, then the check-test is passed successfully, otherwise the test has failed.

Please note that the proposed test sequence also utilizes untested components. In particular these are the read ports of slot 0 and register *r0*. However, it is very unlikely that faults in the read port or in the register file will change the faultless value 0 of *r0* into a -1. This requires, for example, 16 stuck-at-1 faults that are present in *r0*, which is very unlikely.

By considering the binary encoding of the instruction sequence in figure 5-7 (b) it can be noticed that only

- SA0 faults in bits 4 to 7 of the *opc*-field and
- SA0 faults in bits 4 and 5 of the *src1*- and *src2*-field

allow for a correct execution of the test program without being detected, because these bits are always 0 in each instruction of the test program. All stuck-at faults in the fetch and decode register that affect the other bit-fields will be detected.

After executing the test program in figure 5-7 (b) for slot 0, the control is given back to the service core by executing the *halt*-operation. The service core checks bits 0 and 63 of the fault state registers fsr_7 and fsr_8 for the expected values. According to the values of these fault state registers the fault state of a slot is determined as follows:

$$fsSlot(0) := \begin{cases} 1, & \text{if } fsr_7(0) \wedge fsr_8(0) \wedge fsr_7(63) \wedge fsr_8(63) \\ 0, & \text{otherwise} \end{cases}$$

If $fsSlot(0) = 0$, then an adaptation of the remaining test programs must be done. This adaptation is described in more detail in section 5.5. The *check*-test is repeated for the remaining slots of the VARP processor. I.e., the *check*-test for slot $i \in SLOTS - \{0\}$ is done by executing the *check*-operations in figure 5-7 (a) in slot i. This is controlled by the service core that invokes the *check*-test for each slot separately.

5.4.2 Read-Port Test

This section describes the *read port test* of the VARP processor, whose main purpose is the detection of faults in the read ports. But this test program also detects some faults in the pipeline registers that were not covered by the *check*-test and faults in the registers of the register file. The structure of read ports has been already described in section 3.3.2. Please recall that a read port for a register file with 16-bit registers is composed of 16 multiplexer trees as shown in figure 3-15. In [115] minimal test sets were proposed for multiplexer trees. 100 % fault coverage can be obtained for single stuck-at faults with these test sets. A software-based self-test for read ports that rely on such a multiplexer-based structure is presented in [69]. That test distinguishes between data- and address faults. An *address-fault* occurs due to a stuck-at fault in the control signal of a 2:1-multiplexer such that the value from the wrong data input is selected. A *data fault* occurs due to a stuck-at fault in the data input/output of a 2:1-multiplexer. With the test patterns used in [69] all single stuck-at faults are detected with only two test patterns. The test patterns are constructed in such a way that, for both data inputs L and R of a 2:1 multiplexer in a faultless multiplexer tree, it either holds $L = 1$ and $R = 0$ or $L = 0$ and $R = 1$. Figure 5-8 (a) shows a multiplexer tree with 8 data inputs and

the corresponding two test patterns that must be loaded in the registers of the register file. Please recall that a -1 means a bit-vector containing only 1-bits.

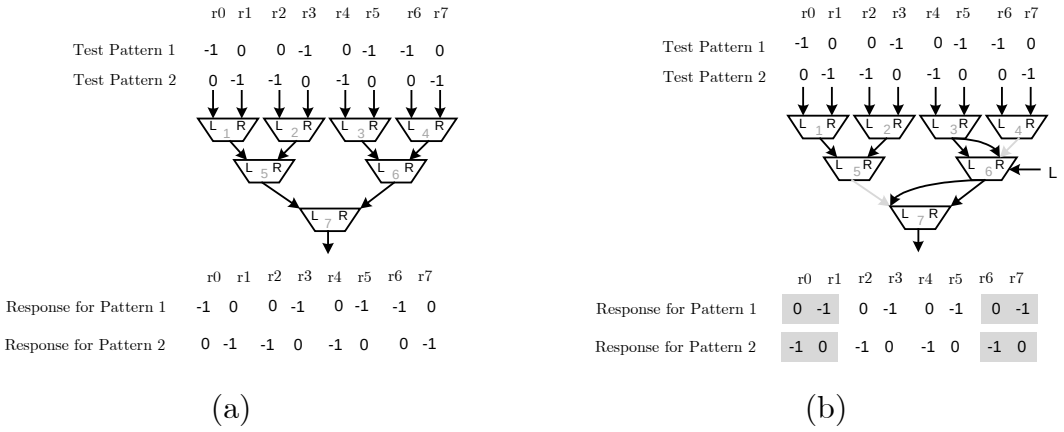


Figure 5-8: (a) Test Patterns and test responses for a faultless 8:1 multiplexer tree according to [69]. (b) Test Patterns and test responses for a 8:1 multiplexer tree with two SA-faults.

The test patterns are applied by loading the registers of the register file with the values shown as test patterns. After that the value of each register is checked by reading the value of each register. If the multiplexer tree is faultless, then the test responses shown in figure 5-8 (a) will be obtained. With these two test patterns, all single address faults and multiple data faults can be detected. Unfortunately, not all multiple address faults can be located correctly. An example is given in figure 5-8 (b) taken from [182]. There the multiplexer tree has two address faults. Multiplexer 6 always selects the left input, and multiplexer 7 always selects the right input, represented by the two additional arrows. As a consequence, only the values of registers *r4* and *r5* can be accessed. The same test response is obtained for a double SA-fault, when multiplexer 5 always selects the right input and multiplexer 7 always selects the left input. Thus, both fault states cannot be distinguished.

In order to detect and localize all multiple address- and data faults correctly, more but simpler test patterns were published in [163, 164] for the VARP processor. These test patterns are listed in table 5-2. Each test pattern defines

- the accessed register r, which is the value of the control signal of the read port, and
- the values for all registers *r0* to *r63*, which are the values at the data inputs of each read port.

No. of Test Pattern	*r*	*r0*	*r1*	*r2*	…	*r62*	*r63*
1	0	0	0	0	…	0	0
2	1	0	0	0	…	0	0
3	2	0	0	0	…	0	0
4	3	0	0	0	…	0	0
…						…	…
64	63	0	0	0	…	0	0
65	0	-1	0	0	…	0	0
66	1	0	-1	0	…	0	0
67	2	0	0	-1		0	0
…						…	…
127	62	0	0	0	...	-1	0
128	63	0	0	0	…	0	-1

Table 5-2: List of test patterns for the read port test.

Test patterns 1 to 64 are used for performing a *stuck-at-1 test* of the read port by setting all data inputs of each multiplexer tree to 0. I.e., all registers are initialized with 0. Each possible register number is applied to *r*. I.e., each register is accessed once with a *chkL-* or *chrR*-operation through each read port. If always a 0 is observed at the outputs of all multiplexer trees, then there is no stuck-at-1 fault on any data input or output of a 2:1 multiplexer that reaches the output of the multiplexer tree. In order to complete the test, a *stuck-at-0 test* of the read port is performed with the test patterns 65 to 128. For each k with $0 \leq k \leq 63$, the data input k of all multiplexer trees is set to 1 while all other data inputs are 0. For this reason, register k is initialized with -1, while all other registers are initialized with 0. Then value k is applied to the control input r of the multiplexer in order to select data input k. If a 1 is observed at each multiplexer-tree output, then input k is read correctly. This is for the following reason: From the stuck-at-1 test of the read port it is known that no stuck-at-1 fault can reach the output of any multiplexer tree. Thus the 1 must be the 1 from input k. It follows immediately that there is also no address fault for selecting input k.

The test patterns from table 5-2 are applied to the read ports of the VARP processor by a test program. The test program[24] that applies the test patterns 1 to 64 is shown in figure 5-9.

[24] Please note that the test code is given for the case that all slots are faultless. If a slot has been detected as faulty by the *check*-test then the read port test program is adapted as described later in the section 5.5.2.

```
// Initialize all registers with zero
ldc 0,r0              nop                   nop                   nop
.                     .                     .                     .
.                     .                     .                     .
.                     .                     .                     .
ldc 0,r63             nop                   nop                   nop
// Fragment 0: SA1 test for all read ports by reading register 0
nop                   nop                   nop                   nop
nop                   nop                   nop                   nop
chkL0 r0,fsr_0,0      chkL0 r0,fsr_2,0      chkL0 r0,fsr_4,0      chkL0 r0,fsr_6,0
nop                   nop                   nop                   nop
nop                   nop                   nop                   nop
chkR0 r0,fsr_1,0      chkR0 r0,fsr_3,0      chkR0 r0,fsr_5,0      chkR0 r0,fsr_7,0
// Fragment 1: SA1 test for all read ports by reading register 1
.                     .                     .                     .
.                     .                     .                     .
.                     .                     .                     .
// Fragment 63: SA1 test for all read ports by reading register 63
chkL0 r63,fsr_0,63    chkL0 r63,fsr_2,63    chkL0 r63,fsr_4,63    chkL0 r63,fsr_6,63
nop                   nop                   nop                   nop
nop                   nop                   nop                   nop
chkR0 r63,fsr_1,63    chkR0 r63,fsr_3,63    chkR0 r63,fsr_5,63    chkR0 r63,fsr_7,63
```

Figure 5-9: Test program for SA1 test in all read ports.

First, the test program initializes all registers with 0. Thereby, the data inputs of all multiplexer trees are set to 0. After that the test program is composed of 64 fragments. The first two NOP-instructions at the beginning of *fragment 0* make sure that the write-back of *r63* is finished before the first instruction with *check*-operations is executed. The two *check*-instructions in each fragment r perform the stuck-at-1 test for the left and right read port of each slot k. If one of these *check*-instructions reads a value different from 0, then the corresponding bit r in the fault state register fsr_k is set to 1. Hence, the following relationship between the values in the fault state registers and the fault state function *fsRP* holds:

$$fsRP(k,r) = \begin{cases} 1, \text{ if } fsr_k(r) = 0 \\ 0, \text{ otherwise} \end{cases}. \tag{5-1}$$

A template for applying the test patterns 65 to 128 from table 5-2 is shown in figure 5-10. This template is parameterized by the register number r from table 5-2 and must be instantiated for each r with $0 \leq r \leq 63$.

```
// All registers are already initialized with zero
ldc #-1,r             nop                   nop                   nop
nop                   nop                   nop                   nop
nop                   nop                   nop                   nop
chkL1 r,fsr0,r        chkL1 r,fsr2,r        chkL1 r,fsr4,r        chkL1 r,fsr6,r
nop                   nop                   nop                   nop
nop                   nop                   nop                   nop
chkR1 r,fsr1,r        chkR1 r,fsr3,r        chkR1 r,fsr5,r        chkR1 r,fsr7,r
ldc #0,r              nop                   nop                   nop
```

Figure 5-10: Test program template for SA0 test of all read port.

First, register r is initialized with -1. The next two NOP-instructions make sure that the -1 is written back to the register file before the value is accessed through the read-port. When the subsequent *chkL1*- and *chkR1*-instructions are executed, then only data input r of all read ports is -1. Thus, the *chkL1*- and *chkR1*-instructions should read the value -1. If this is not the case for read port k, then bit r of fsr_k is set to 1, such that again the relationship from equation (5-1) holds.

It should be noted that also the values from the pipeline registers must be taken into account for the read port test. This is due to the fact that the output of the read port is used as input of the multiplexer-tree of the bypass (see figure 3-17). I.e, the pipeline registers must be all 0 for the read port test, because the outputs of these pipeline registers are connected to the inputs *ex1*, ..., *ex4*, *wb1*, ..., *wb4* of the multiplexer trees of each bypass (see figure 3-17). The test code in figure 5-9 and figure 5-10 already fulfills this requirement, because, by the two NOP-instructions before each *check*-instruction, it is ensured that all pipeline registers have the value 0, when the *check*-instruction is executed.

The proposed test patterns in table 5-2 and the corresponding test program in figure 5-10 were developed manually. In order to determine the fault coverage and to prove the diagnostic capability, an exhaustive fault simulation was done for the VARP processor in [148] by using the stuck-at fault model. I.e., a VHDL model of the VARP processor with a structural implementation of the read port was used for the fault simulation. The read port was implemented by using 2:1-multiplexers. In total 6092 faults exist in the read port. Each one of these faults was injected in the VHDL model, and the test programs from figure 5-9 and figure 5-10 were executed by a simulation in the VHDL model. It turned out that each fault could be detected. Moreover, each fault has been classified correctly by the test program regarding the fault state of the processor. The same experiment was done for all double faults in the read ports [148]. In this case, also 100% fault coverage could be obtained, including the correct diagnosis of all double faults.

5.4.3 Slot-Test

The slot-test encapsulates the *check*-test and the read port test in order to localize faults in the pipeline registers that could not be detected with the *check*-test. In particular these are faults that prevent the *ldc*-operation, which is extensively used for initializing registers during the read port test, from being executed correctly. In particular, the *check*-test cannot detect stuck-at-0 faults that affect the gray shaded bit-fields of the *opcode*-, *src1*-, and *src2*-bit fields shown in figure 5-7. Stuck-at-0 faults in these bit fields will

- change the opcode of a *ldc*-operation into the opcode of another operation, or
- the specified constant of the *ldc*-operation is changed.
- Moreover, due to stuck-at faults in some bit-fields of the write-back register of a slot, the address of the destination register may be changed, too.

In any case, a specified destination register r of the *ldc*-operations in figure 5-10 is not initialized correctly with -1. As a consequence, the *check*-operations for each read-port k that access register r will detect a fault such that the property

$fsRP(k,r) = 0$ holds for all $k \in RPS$. I.e., a fault state is detected, where each read port has a fault, when accessing register r. However, in such situations, it is more likely that register r is faulty or initialized incorrectly than having the same fault in all read ports. For this reason the service core checks for such suspicious fault states. In order to do this in an efficient way, the fault state *fsRF* of the register file is used. When the read port test is finished, the fault state *fsRF* of the register file is set by the service core as follows:

$$fsRF(r) := \begin{cases} 0, \text{ if } \forall k \in RPS : fsRP(k,r) = 0 \\ 1, \text{ otherwise} \end{cases} \quad (5\text{-}2)$$

Based on the fault state *fsRF* of the register file, it is decided by the service core for what reason the read port test has failed. This can be either a fault in the read port, a fault in a register of the register file, or a fault in a pipeline register of that slot that has executed the initialization code. These three cases are distinguished by the *syndrome* obtained from the results of the function *fsRF*. The syndrome is considered as a bit-vector (*fsRF*(0), *fsRF*(1),..., *fsRF*(63)). Faults in the pipeline registers of the slot that has executed the initialization code for the read port test are recognized by the syndrome-patterns shown in table 5-3. Please note that the service core recognizes a pattern, when the 0-bits of the pattern in table 5-3 will match with the 0-bits of the syndrome. I.e., a 1 in the pattern is interpreted as a don't care bit in the syndrome. If the service core recognizes one of these patterns, then the slot that has executed the initialization code (i.e., the *ldc*-operations) is declared as faulty.

Syndrome patterns	**Fault**	
	Type	**SA**
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000	Data	0/1
10101010 10101010 10101010 10101010 10101010 10101010 10101010 10101010	Adr0	0
01010101 01010101 01010101 01010101 01010101 01010101 01010101 01010101		1
11001100 11001100 11001100 11001100 11001100 11001100 11001100 11001100	Adr1	0
00110011 00110011 00110011 00110011 00110011 00110011 00110011 00110011		1
11110000 11110000 11110000 11110000 11110000 11110000 11110000 11110000	Adr2	0
00001111 00001111 00001111 00001111 00001111 00001111 00001111 00001111		1
11111111 00000000 11111111 00000000 11111111 00000000 11111111 00000000	Adr3	0
00000000 11111111 00000000 11111111 00000000 11111111 00000000 11111111		1
11111111 11111111 00000000 00000000 11111111 11111111 00000000 00000000	Adr4	0
00000000 00000000 11111111 11111111 00000000 00000000 11111111 11111111		1
11111111 11111111 11111111 11111111 00000000 00000000 00000000 00000000	Adr5	0
00000000 00000000 00000000 00000000 11111111 11111111 11111111 11111111		1

Table 5-3: Patterns for *fsRF* used for recognizing faults in pipeline registers.

For example, the first pattern with type *data* occurs, when the destination register r is always initialized with a wrong value. This may happen due to a stuck-at-0

fault in the *src1*- and *src2*-bit fields of the pipeline registers. Then, reading of each register through each read port fails, such that each register is declared as faulty. The remaining syndromes of type Adr$x$ occur due to stuck-at-0 or stuck-at-1 faults that affect bit x in the destination register bit-field of the write-back register. For example, if bit 0 ($x = 0$) in the destination register bit-field of the write-back register in slot 0 has a SA1-fault, then the destination register address is always changed to an odd address. Thus, when accessing registers with an even address during the read port test, a wrong value is obtained, and registers with even address are declared as faulty.

Stuck-at faults in a register r of the register file will be recognized by syndromes that do not match with any pattern in table 5-3, but $fsRF(r) = 0$, i.e., register r could not be accessed correctly through any read port. Please recall that the read port test stores 0 as well as -1 in each register and will check for each value if it can be accessed correctly through each read port. If the slot cannot write correctly in a particular set of processor registers $\{r_1,\ldots,r_k\}$ or there is a stuck-at fault in registers $\{r_1,\ldots,r_k\}$, then the read port test will detect a fault in each read port, when reading a register $r \in \{r_1,\ldots,r_k\}$. For $k < 32$ this will result in a syndrome, different from the syndromes listed in table 5-3, where $fsRF(r) = 0$ if and only if $r \in \{r_1,\ldots,r_k\}$. For such syndromes the registers $\{r_1,\ldots,r_k\}$ are declared as faulty.

The read port test programs shown in figure 5-9 and figure 5-10 will detect only faults in the pipeline registers of slot 0, because the initialization code is executed only in slot 0. For this reason, the slot test includes the execution of four versions of the read port test. Version i executes all initialization code (i.e., all *ldc*-operations) in slot i. The slot test starts with $i = 0$. After executing version i of the read port test, the control is given back to the service core that evaluates the syndromes obtained from the fault state function *fsRF*, and, if necessary, declares slot i as faulty. If slot i is declared as faulty, then the fault state *fsRP* of the read ports and *fsRF* are reset by the service core before version i+1 of the read port test is invoked. Otherwise the fault state *fsRP* determined for the read ports by the read port test is accepted. In order to save program memory, the service core modifies version i of the read port test by moving the initialization code from slot i into slot i+1 before the read port test is started again.

5.4.4 Bypass Test

As it was already shown in figure 3-17, the structure of the bypass is almost the same as the structure of the read port. Moreover, bypass and read-port are connected in such a way that they form together a multiplexer tree again (see figure 3-17). I.e. the output of a read port is used as data input for the bypass (see figure 5-11) that selects between the inputs from the pipeline registers

(*ex0*,...,*ex3*,*wb0*,...,*wb3*) and the input from the *read port*. Obviously, the whole read port and bypass structure is equivalent to the structure of a read port. Therefore, the organization of the bypass test is very similar to the organization of the read-port test. However, the generation of the address signal for the multiplexer tree of the bypass is more complex.

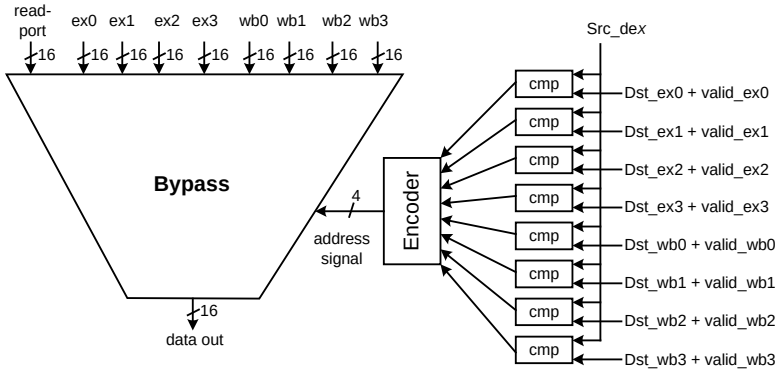


Figure 5-11: Bypass with bypass control logic.

The address signal is used to select either the input from the read port (no forwarding) or from one of the pipeline registers (forwarding). The address signal is generated by an encoder, and the input signals for the encoder are generated by compare-components (*cmp*). Each *cmp* compares the numbers of a destination register from the write-back- and execute-stage (*Dst_ex0*,...,*Dst_wb3*, *Dst_wb0*,...,*Dst_wb3*) with the used source register number in the decode stage (*Src_dex*) with respect to the valid-flag of each pipeline register. Encoder and *cmp*-components can be considered as the control logic for the bypass. Two sub-functionalities of the bypass are tested separately:

- First, the forwarding of values from pipeline registers is done correctly.
- Second, the read port value is selected correctly, if no forwarding is needed.

The first sub-functionality of the bypass is tested by applying the test patterns shown in table 5-4 at the inputs shown in figure 5-11 by using an appropriate instruction sequence.

No.	data inputs									Dst_								src_de
	read port	ex0	ex1	ex2	ex3	wb0	wb1	wb2	wb3	ex0	ex1	ex2	ex3	wb0	wb1	wb2	wb3	
1	0	0	0	0	0	0	0	0	0	0	-	-	-	-	-	-	-	0
2	0	0	0	0	0	0	0	0	0	-	0	-	-	-	-	-	-	0
...																		...
8	0	0	0	0	0	0	0	0	0	-	-	-	-	-	-	-	0	0
9	0	0	0	0	0	0	0	0	0	63	-	-	-	-	-	-	-	63
10	0	0	0	0	0	0	0	0	0	-	63	-	-	-	-	-	-	63
...																		...
16	0	0	0	0	0	0	0	0	0	-	-	-	-	-	-	-	63	63
17	0	-1	0	0	0	0	0	0	0	0	-	-	-	-	-	-	-	0
18	0	0	-1	0	0	0	0	0	0	-	0	-	-	-	-	-	-	0
...																		...
24	0	0	0	0	0	0	0	0	-1	-	-	-	-	-	-	-	0	0
25	0	-1	0	0	0	0	0	0	0	63	-	-	-	-	-	-	-	63
26	0	0	-1	0	0	0	0	0	0	-	63	-	-	-	-	-	-	63
...																		...
32	0	0	0	0	0	0	0	0	-1	-	-	-	-	-	-	-	63	63

Table 5-4: Test patterns for testing the forwarding-functionality of a bypass.

A – in columns *Dst_ ex0*,...,*Dst_ wb3* means that the value of the corresponding pipeline register is not valid. This is achieved by executing a NOP. Please note that the execution of a NOP computes the value 0 at the output of the execution unit, such that the corresponding data input of the bypass is 0. Moreover, the bit-field for the destination register number in a pipeline register is set to 0 by a NOP. Test patterns 1 to 16 check for stuck-at-1 faults in the multiplexer trees of the bypass, in the same manner as it was done for the read ports by the stuck-at-1 test. I.e., all of the test patterns 1 to 16 set all data inputs of the bypass to 0. Moreover, each of the test patterns forces the bypass to select another data input. The obtained output value is compared with 0. This is accomplished in two phases in order test the control logic of the bypass, too. The first phase (1-8) uses register 0 as source register in the decode stage ($src\_de = 0$). The second phase (9-16) uses register 63 as source register in the decode stage ($src\_de = 63$). By using these register numbers, all inputs of a single *cmp*-component (see figure 5-11) are set to 0 by the test patterns 1 to 8, and they are set to 1 by the test patterns 9 to 16.

Test patterns 17 to 32 perform the stuck-at-0 test in the same manner as it was done during the read port test. The test is done in two phases again (17-32), where the first phase uses register *r0* and the second phase uses register *r63* for testing the selection of correct data input of the bypass. Please recall that during the stuck-at-0 test only the selected data input of the bypass is set to -1, while all

other data inputs are set to 0. By this also address faults in the multiplexer tree will be detected.

In order to apply the presented test patterns with a test program to the bypasses, the test program is composed of (d,k,p)-instruction groups, where $d \in \{\text{L,R}\}$, $k \in SLOTS$, and $p \in \{1,2\}$. A (d,k,p)-instruction group executes an *ldc*-operation in slot k. p instructions later all slots execute either *chkL*-operations ($d = \text{L}$) or *chkR*-operations ($d = \text{R}$). Hence, for $p = 1$, the constant value of the *ldc*-operation is forwarded from the execution stage of slot k into the decode stages of all slots. For $p = 2$, the constant value of the *ldc*-operation is forwarded from the write-back stage of slot k into the decode stages of all slots. Figure 5-12 shows a code fragment of the bypass test. This code fragment is used for testing the forwarding of a value from slot 1 through all left bypasses. The applied test patterns from table 5-4 are referenced in the listing.

```
// SA1 test with source/destination register 0; applies test patterns 2 and 6
nop                     nop                     nop                 nop
nop                     ldc #0,r0               nop                 nop
chkL0 r0,fsr8,0         chkL0 r0,fsr8,1         chkL0 r0,fsr8,2     chkL0 r0,fsr8,3
nop                     ldc #0,r0               nop                 nop
nop                     nop                     nop                 nop;
chkL0 r0,fsr8,4         chkL0 r0,fsr8,5         chkL0 r0,fsr8,6     chkL0 r0,fsr8,7
// SA0 test with source/destination register 0; applies test patterns 18 and 22
nop                     nop                     nop                 nop
nop                     ldc #-1,r0              nop                 nop
chkL1 r0,fsr8,0         chkL1 r0,fsr8,1         chkL1 r0,fsr8,2     chkL1 r0,fsr8,3
nop                     ldc #0,r0               nop                 nop
nop                     ldc #-1,r0              nop                 nop
nop                     nop                     nop                 nop
chkL1 r0,fsr8,4         chkL1 r0,fsr8,5         chkL1 r0,fsr8,6     chkL1 r0,fsr8,7
// SA1 test with source/destination register 63; applies test patterns 10 and 14
nop                     nop                     nop                 nop
nop                     ldc #0,r63              nop                 nop
chkL0 r63,fsr8,0        chkL0 r63,fsr8,1        chkL0 r63,fsr8,2    chkL0 r63,fsr8,3
nop                     ldc #0,r63              nop                 nop
nop                     nop                     nop                 nop;
chkL0 r63,fsr8,4        chkL0 r63,fsr8,5        chkL0 r63,fsr8,6    chkL0 r63,fsr8,7
// SA0 test with source/destination register 63; applies test patterns 26 and 30
nop                     nop                     nop                 nop
nop                     ldc #-1,r63             nop                 nop
chkL1 r63,fsr8,0        chkL1 r63,fsr8,1        chkL1 r63,fsr8,2    chkL1 r63,fsr8,3
nop                     ldc #0,r63              nop                 nop
nop                     ldc #-1,r63             nop                 nop
nop                     nop                     nop                 nop
chkL1 r63,fsr8,4        chkL1 r63,fsr8,5        chkL1 r63,fsr8,6    chkL1 r63,fsr8,7
```

Figure 5-12: Test program for testing forwarding from slot 1.

For the test program in figure 5-12 it is assumed that all registers of the register file are initialized with 0 such that the read port input of the bypass has the value 0. The first three instructions form a (L,1,1)-instruction group. The second instruction executes the *ldc*-operation, whose value should be forwarded. When the third instruction enters the decode-stage, then the *ldc*-operation enters the execution stage, and instruction 1 is in the write-back stage such that all data inputs of the bypasses have the value 0. Because the third instruction is in the decode stage, all for slots decode a *chkL0*-operation. I.e., the left bypass of each slot must select the value written into *r0* by the *ldc*-operation from instruction 2. Hence, all four left bypasses select data input *ex1* (see figure 5-11). Instructions 4 to 6 form a (L,1,2)-instruction group and force the left bypasses to select the value of the *ldc*-operation from the write-back stage. Instructions 7 to 13 perform the

same test sequence as instructions 1 to 6, but they perform the stuck-at-0 test by loading constant -1 into register *r0*. Finally, the same test is repeated for the second phase, i.e., register *r63* is used for forcing the bypass to perform forwarding from execution- and write-back stage. Hence, the instruction sequence in figure 5-12 checks for all left bypasses of the VARP processor that they perform forwarding from the execute- and write-back stage of slot 1. Similar test programs are needed for testing the right bypasses in each slot, and for testing forwarding from other slots than slot 1.

The fault coverage and diagnostic capability of this bypass test program was evaluated by a fault simulation in [148]. For this reason a structural implementation of the bypass and the bypass control logic was integrated into the VARP processor model. The structural bypass model has 1278 stuck-at faults. A fault simulation was done for each single-stuck-at fault in the bypass. I.e., the described bypass test program has been executed in the VHDL model with the injected fault. For the bypass a fault-coverage of 91.0% was obtained. By a manual analysis of the missed faults it was discovered that these faults are located in the control logic of the bypass. These were exactly those faults that will cause a false-forwarding, i.e., a value from a pipeline register is selected, although the bypass must select the read port value. By a detailed analysis of the bypass control logic, test patterns for exciting these faults were determined manually. These test patterns must have the property that the binary register number r of the source register read in the decode stage is distinct in exactly one bit position from the binary register number r' of the destination registers written in the execute- and write-back stage. In that situation a forwarding should not take place. However, if there is a stuck-at fault in the bypass control such that the single bit position where r and r' differ becomes equal, then a false forwarding takes place and can be detected. A template for a test program that will check for this false-forwarding is shown in figure 5-13. The template is parameterized with r and r'. The first group of instructions checks for a false-forwarding from the execution stage into the decode stage. It is assumed that all registers of the register file are initialized with 0. When a false forwarding takes place, then a *chkL0*-operation from the first instruction group reads the -1 from one of the preceding *ldc*-operations. In a similar way the remaining three instruction groups will check for a false forwarding from the write-back stage into the decode-stage through the left bypass, and for a false forwarding through the right bypasses.

```
// Group 1: Check left bypasses for false-forwarding from EX-stage
nop                     nop                     nop                     nop
ldc #-1,r'              ldc #-1,r'              ldc #-1,r'              ldc #-1,r'
chkL0 r,fsr8,0          chkL0 r,fsr8,2          chkL0 r,fsr8,4          chkL0 r,fsr8,6
ldc #0,r                ldc #0,r'               nop                     nop
nop                     nop                     nop                     nop;
nop                     nop                     nop                     nop;
// Group 2: Check left bypasses for false-forwarding from WB-stage
ldc #-1,r'              ldc #-1,r'              ldc #-1,r'              ldc #-1,r'
nop                     nop                     nop                     nop;
chkL0 r,fsr8,0          chkL0 r,fsr8,2          chkL0 r,fsr8,4          chkL0 r,fsr8,6
ldc #0,r                ldc #0,r'               nop                     nop
nop                     nop                     nop                     nop;
nop                     nop                     nop                     nop;
// Group 3: Check right bypasses for false-forwarding from EX-stage
ldc #-1,r'              ldc #-1,r'              ldc #-1,r'              ldc #-1,r'
chkR0 r,fsr8,1          chkR0 r,fsr8,3          chkR0 r,fsr8,5          chkR0 r,fsr8,7
ldc #0,r                ldc #0,r'               nop                     nop
nop                     nop                     nop                     nop;
nop                     nop                     nop                     nop;
// Group 4: Check right bypasses for false-forwarding from WB-stage
ldc #-1,r'              ldc #-1,r'              ldc #-1,r'              ldc #-1,r'
nop                     nop                     nop                     nop;
chkR0 r,fsr8,1          chkR0 r,fsr8,3          chkR0 r,fsr8,5          chkR0 r,fsr8,7
ldc #0,r                ldc #0,r'               nop                     nop
```

Figure 5-13: Test program template for testing for false-forwarding.

Thereby, the template must be instantiated for the combinations of register addresses r and r' shown in table 5-5.

Instance	r	r'
1	000000	000001
2	000000	000010
3	000000	000100
4	000000	001000
5	000000	010000
6	000000	100000
7	111111	111110
8	111111	111101
9	111111	111011
10	111111	110111
11	111111	101111
12	111111	011111

Table 5-5: Register combinations used for the false-forwarding test.

If a false forwarding is detected by the false-forwarding test program for bypass k, then it cannot be guaranteed that bypass k provides correct values from the read port k. For this reason such a bypass fault is considered as equivalent to a fault in the read port k, where the access to each register is faulty. Thus, $fsRP(k,r) \coloneqq 0$ for all $r \in \text{REGS}$, if bit k in fsr_8 is set to one by the false-forwarding test program from figure 5-13.

Performing a fault simulation with the test program derived from the template in figure 5-13 results in a fault-coverage of 55% for a bypass. In particular this test program covers all of those faults that were not covered from the test program for the forwarding-test in figure 5-12. Thus, by executing both test programs, a fault coverage of 100% is obtained for the bypasses including the bypass control logic.

5.4.5 False-Write-Back Test

With the read port test it was also tested that a value was written correctly into a specified destination register. But it is not checked whether or not the values of other registers than the destination register may be affected by the write operation. This may happen due to the structure of a register as it is shown in figure 5-14.

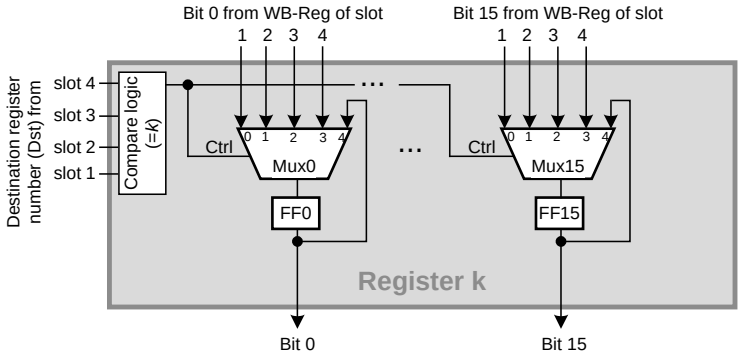


Figure 5-14: Structure of a single register.

A 16-bit register in the register file consists of 16 flip-flops. Each flip-flop receives its input from a 5:1-multiplexer. Thereby, data input m of each multiplexer n is connected with bit n of the write-back register in slot m. Moreover, data input 4 is connected with the output of the flip-flop. This data input is used to hold the current value of the flip-flop. The compare logic of register k is responsible for generating the control signal $ctrl_k$ for the multiplexers, and it receives the destination register numbers *dst0*,…,*dst3* from the write-back stage of all slots:

$$ctrl_k(dst0, ds1, dst2, dst3) := \begin{cases} 0, \text{ if } dst0 = k \\ 1, \text{ if } dst1 = k \\ 2, \text{ if } dst2 = k \\ 3, \text{ if } dst3 = k \\ 4, \text{ otherwise} \end{cases}$$

All multiplexers receive the same control signal. A fault in the control logic or an address fault in a 5:1-multiplexer may cause loading a new value into register k, although register k should hold its current value. In order to check whether or not a register, which is not the destination register of an operation, is affected by a write operation, all registers are initialized with 0. Then the value -1 is written into a single register r. After that the value of all other registers is compared with 0. If the value of one of these registers is different from 0, then it is declared as faulty. Figure 5-15 shows the template for this test program.

```
// Initialize all registers with 0
ldc #0, r0              nop                 nop                 nop
...
ldc #0, r63             nop                 nop                 nop
// Write test value -1 into register r
ldc #-1,r;              ldc #-1,r;          ldc #-1,r;          ldc #-1,r
nop;                    nop;                nop;                nop
nop;                    nop;                nop;                nop
chkL0 r0,fsr9,0;        nop;                nop;                nop
...
chkL0 r-1,fsr9,r-1;     nop;                nop;                nop
chkL0 r+1,fsr9,r+1;     nop;                nop;                nop
...
chkL0 r63,fsr9,63;      nop;                nop;                nop
```

Figure 5-15: Test program template for the false-write-back test.

The test program template initializes all registers with 0 and then stores -1 in register r. Please note that this operation is executed in all four slots. By this, all registers in the register file have a 1-bit at all data inputs 0 to 3 of their multiplexers (see figure 5-14). If one of the registers accidently loads another value than its own value at data input 4, then it will be a 1-bit. The *check*-operations in figure 5-14 will check all registers, except register r, to have the value 0. If one of them is not 0 (e.g. register j), then it was loaded accidently with a 1-bit. Thus bit j in fault state register 9 is set to 1, indicating that register j cannot be used as destination register, because writing to register r overwrites the value in register j.

This template in figure 5-15 must be instantiated for each register $r \in$ REGS. For practical reasons, the template is not instantiated statically 64 times. Rather the service core generates dynamically the appropriate instances. I.e., originally only a single instance of the template exists for $r = 0$. After executing this instance, the service core updates the fault state of each register, adapts the test program for r in such a way that the test program for r+1 is obtained, and the adapted test program is started again.

In [148] the register structure shown in figure 5-15 was implemented for register *r0* with simple gates. This structural model has in total 566 faults and was included in the model of the VARP processor. Each of the 566 faults was injected in the structural model before executing the slot test program and the false-write-back test program. Both test programs together detected 100% of the faults, including a correct classification of the caused functional misbehavior.

5.4.6 Execution Unit Test

The generation of test programs for the execution units can be done straightforward with ATPG-tool support. Appropriate methods were described for example in [181] and [163]. The same techniques were used in [81] for the generation of test programs for a 16-bit execution unit that is very similar to the one of the VARP processor. I.e., an ATPG tool was used for test pattern generation for an execution unit. The operand values for a test program template were extracted from the generated test patterns, and they were used to fill the

gaps in a test program template. In order to generate a test for a specific operation type, the control signals of the execution unit can be fixed by some ATPG constraints during test pattern generation. For the execution units a fault coverage of 99.93% could be obtained with the generated test patterns, requiring approximately 4.3 kByte of memory for storing the test sequences [81].

The generated test programs will have the same structure as the example shown for the *adder*-test in figure 5-2. Therefore, the test patterns can be applied to an execution unit in a straightforward manner, because the data transportation paths of the VARP processor needed for register initialization and capturing were already tested with the previous test programs. By knowing the faults on these paths, correct operand values can be provided to each execution unit for test purposes by adapting the test programs for the execution units in the same way as it was sketched for the *adder*-test in figure 5-3. For those reasons, and because the software-based self-test of simple ALUs is well established [141], the execution unit test programs and their adaptation was not further considered for the implementation of the adaptive SBST. Rather, the more complex adaptation of the test programs, which are used for testing the data transportation in the data path, is described in the subsequent section.

5.5 Adapting the Test Programs

The previous section 5.4 has described in detail the test programs used for testing particular components of the VARP processor. In the subsequent sections it is described how the service core adapts these self-test programs when faults are detected. Please recall that each test program returns control to the service core by executing the *halt*-operation. When the service core gets the control back, it can check the fault state registers and adapts the test programs to the current fault state of the processor.

5.5.1 Adaptation of the Check-Test

The execution of the check-test from figure 5-7 starts for slot 0, and it is repeated for slots 1 to 3. After executing the check-test for slot k, the service core exchanges in each instruction the operation from slot k with the NOP from slot $(k{+}1) \bmod 4$ before the check-test is started again for slot $k{+}1$. Please note that this adaptation is not done for adapting the test program to the current fault state of the processor. Rather it is only done for saving program memory. By this the check-test program requires only 5 instructions instead of 20. In any case, the read port test is started after executing the check-test for each slot.

5.5.2 Adaptation of the Read Port Test

The read port test program must be adapted to fault states that were detected by the check-test and by the slot-test. Please note that the read port test is part of the slot test, i.e., various versions of the read port test program are executed during the slot test. Thereby each version must be adapted to fault states detected by previously executed versions of the read port test.

5.5.2.1 Adaptations due to the Check-Test

If the check-tests have detected faults in slots $k_1,\ldots,k_m$, then these slots are declared as faulty, and they will be only used for executing NOPs. Hence, there is no need for testing the read ports of these slots. Moreover, they cannot be used for executing the initialization code for the read port test of other slots. For this reason some versions of the read port test are skipped. In particular, only the versions $k \in SLOTS - \{k_1,\ldots,k_m\}$ of the read port test will be executed. Please recall that for version k of the read port test program all *ldc*-operations are moved into slot k. Moreover, all *check*-operations in slots $k_1,\ldots,k_m$ are replaced by NOPs. This must be done due to the fact that the operands of a *check*-operation may be altered by the faults detected during the check-test, and, therefore, the *check*-operation would set a wrong bit in the fault state registers. In figure 5-16 it is shown how the template from figure 5-10 is adapted to faults in slots 0 and 2. The *ldc*-operations from slot 0 were moved to slot 1 and the *chk*-operations in slot 0 and slot 2 were replaced by NOPs.

```
// All registers are already initialized with zero
nop                  ldc #-1,r            nop                  nop
nop                  nop                  nop                  nop
nop                  nop                  nop                  nop
nop                  chkL1 r,fsr2,r       nop                  chkL1 r,fsr6,r
nop                  nop                  nop                  nop
nop                  nop                  nop                  nop
nop                  chkR1 r,fsr3,r       nop                  chkR1 r,fsr7,r
nop                  nop                  nop                  nop
```

Figure 5-16: Test program template from figure 5-10 adapted to faults in slots 0 and 2.

5.5.2.2 Adaptation due to the Slot-Test

Now suppose the slot test has detected some faults in the pipeline registers of slot k. This happens when the slot test recognizes the syndromes shown in Table 5-3 after executing version k of the read port test program. In this case

- slot k is declared as faulty,
- the fault state of the read ports is restored to the previously known fault state,
- *check*-operations executed in slot k are replaced by NOPs, and
- *ldc*-operations are moved into the next faultless slot $k' > k$.

Faults detected from a read port test in a register or in a read port will not cause any adaptation of the read port test.

5.5.3 Adaptation of the Bypass Test

The bypass test must be adapted only to slot faults. An adaptation of the bypass test due to a detected fault in a read port is not needed, because the value provided through the read port is not used from the bypass. If the read port test has detected a faulty register, then the bypass test is not adapted, because the bypass test does not use the value from the register. Rather, it uses the register address only for address generation in the bypass control logic. Thus, even then, when a fault in registers *r0* or *r63* is detected, the bypass test still uses these registers, in order to test for stuck-at faults in the address lines of the bypass.

5.5.3.1 Adaptation due to the Slot-Test

Suppose that a slot k was detected as faulty by the slot test. The bypass forwarding test shown in figure 5-12 is adapted as follows:

- First, all (d,k,p)-instruction groups are deleted from the test program for each $d \in \{L,R\}$ and each $p \in \{1,2\}$.
- Second, in all (d,k',p)-instruction groups, where $k' \neq k$, the *check*-operations executed in slot k are replaced by NOPs

As an example, consider the portion of the bypass test program shown in figure 5-12. This test program is composed of 8 instruction groups. Each group executes an *ldc*-operation in slot 1. Thus, when slot 1 is faulty, then the whole portion of the test program can be removed, because slot 1 will not generate data that is accessed by other slots through their bypasses. However, when slot 2 will be faulty, then all *check*-operations in slot 2 must be replaced by NOPs. By this the bypass of slot 2 is not tested anymore. The adapted test program for the latter case is shown in figure 5-17.

The false-forwarding test program, based on the template shown in figure 5-13, is adapted in a similar way. When slot k is faulty, then all operations in the faulty slot k are replaced by NOPs. By this, the bypasses of faultless slots are tested for false-forwarding, but under the same condition as the user application is running later, i.e., all faulty slots execute NOPs.

```
// SA1 test with source/destination register 0; applies test patterns 2 and 6
nop                     nop                     nop                 nop
nop                     ldc #0,r0               nop                 nop
chkL0 r0,fsr8,0         chkL0 r0,fsr8,1         nop                 chkL0 r0,fsr8,3
nop                     ldc #0,r0               nop                 nop
nop                     nop                     nop                 nop;
chkL0 r0,fsr8,4         chkL0 r0,fsr8,5         nop                 chkL0 r0,fsr8,7
// SA0 test with source/destination register 0; applies test patterns 18 and 22
nop                     nop                     nop                 nop
nop                     ldc #-1,r0              nop                 nop
chkL1 r0,fsr8,0         chkL1 r0,fsr8,1         nop                 chkL1 r0,fsr8,3
nop                     ldc #0,r0               nop                 nop
nop                     ldc #-1,r0              nop                 nop
nop                     nop                     nop                 nop
chkL1 r0,fsr8,4         chkL1 r0,fsr8,5         nop                 chkL1 r0,fsr8,7
// SA1 test with source/destination register 63; applies test patterns 10 and 14
nop                     nop                     nop                 nop
nop                     ldc #0,r63              nop                 nop
chkL0 r63,fsr8,0        chkL0 r63,fsr8,1        nop                 chkL0 r63,fsr8,3
nop                     ldc #0,r63              nop                 nop
nop                     nop                     nop                 nop;
chkL0 r63,fsr8,4        chkL0 r63,fsr8,5        nop                 chkL0 r63,fsr8,7
// SA0 test with source/destination register 63; applies test patterns 26 and 30
nop                     nop                     nop                 nop
nop                     ldc #-1,r63             nop                 nop
chkL1 r63,fsr8,0        chkL1 r63,fsr8,1        nop                 chkL1 r63,fsr8,3
nop                     ldc #0,r63              nop                 nop
nop                     ldc #-1,r63             nop                 nop
nop                     nop                     nop                 nop
chkL1 r63,fsr8,4        chkL1 r63,fsr8,5        nop                 chkL1 r63,fsr8,7
```

Figure 5-17: Adapted test program from figure 5-12 for testing forwarding from slot 1, when slot 2 is faulty.

5.5.4 Adaptation of the False-Write-Back Test

The false-write-back test shown in Figure 5-15 must be adapted to slot-faults, faults in the read ports, faults in the registers, and faults in the bypasses.

5.5.4.1 Adaptations due to the Slot Test

When a faulty slot was detected during the slot test, then all *ldc*- and *check*-operations in the false-write-back test program are moved into a faultless slot, except the *ldc #-1,r* operation in a faulty slot. This operation is deleted.

5.5.4.2 Adaptations due to the Read Port and Bypass Test

For the false-write-back test program it is important to execute each *check*-operation in a slot k, where the accessed register r is read correctly, i.e., either $fsRP(2 \cdot k, r) = 1$ or $fsRP(2 \cdot k + 1, r) = 1$. For this reason a *chkL*-operation may be changed into a *chkR*-operation or the *check*-operation is moved into another slot k'. Please note that this decision is made individually for each *check*-operation.

Faults in the bypasses that were detected from the forwarding test program cannot affect the false-write-back test program, because the test program does not utilizes the forwarding functionality of the bypass. But faults that were detected from the false-forwarding test program may prevent the *check*-operation from reading the value from the register file. However, a fault detected from the false-forwarding test program for bypass p, is equivalent to a fault of the read port p, where no register can be accessed correctly through read port p. For this reason such faults are already handled as described above by moving *check*-operations into other slots or changing *chkL*-operations into *chkR*-operations.

If the read port test has detected a faulty register, then the corresponding *check*-operation is deleted in the test program. As an example consider the adapted false-write-back test program in figure 5-18. There it is assumed that slot 0 is faulty, and registers *r0* to *r3* cannot be accessed correctly through the left read port of slot 1.

```
// Initialize all registers with 0
nop                 ldc #0, r0            nop                 nop
...
nop                 ldc #0, r63           nop                 nop
// Write test value -1 into register r
nop                 ldc #-1,r             ldc #-1,r           ldc #-1,r
nop                 nop                   nop                 nop
nop                 nop                   nop                 nop
nop                 nop                   chkL0 r0,fsr9,0     nop
nop                 nop                   chkL0 r1,fsr9,1     nop
nop                 nop                   chkL0 r2,fsr9,2     nop
nop                 nop                   chkL0 r3,fsr9,3     nop
nop                 chkL0 r4,fsr9,4       nop                 nop
...                 ...                   ...                 ...
nop                 chkL0 r63,fsr9,63     nop                 nop
```

Figure 5-18: Example for an adapted false-write-back test program.

5.6 Results

In [148] the proposed self-test programs were evaluated using a gate-level implementation of the tested components. An integration of these self-test programs into the system presented in figure 5-5 together with the software-based rescheduling algorithm, including an analysis of the diagnostic capabilities was done in [153]. The results of both works are presented in the subsequent sections.

5.6.1 Hardware Overhead

The system shown in figure 5-5 contains various components for administrating the software-based self-test. In table 5-6 the sizes of these hardware extensions are reported.

Component	Cell area in µm²	Size in % of the VARP processor
Arbiter	1941	7.2%
Service Core	1341	5.0%
VARP-extensions	536	2.0%

Table 5-6: Hardware-Overhead for the administrative components of the adaptive software-based self-test.

The arbiter in that system configuration becomes bigger than the arbiter presented in section 3.2.2 for the software-based self-repair, because program memory access of the service core must be managed. The service core is a small 8-bit processor with an application specific instruction set that is adopted to the tasks of managing the adaptive self-repair process. The size of this service core is comparable with the size of a single read port of the VARP processor. The VARP-extensions reported in table 5-6 include the additional hardware needed for the implementation of the *check*- and *halt*-operations in each slot. Please note that

most of the presented hardware overhead is not needed, when this approach is used in a multi-core system with shared program and data memory. Such a system was presented in [124]. Each core of the system may be used as a service core during the startup phase, and due to the shared memory, the arbiter will become superfluous, too.

5.6.2 Memory Consumption

Beside the hardware overhead, also additional memory is needed for storing the self-test programs and the service routine of the service core. Table 5-7 reports this memory consumption.

Service Core Memory		
Administration of	*Instructions*	*Bytes*
check-test	135	135
read port test	659	659
bypass test	80	80
false-write-back test	487	487
rescheduling algorithm	336	336
miscellaneous data	229	229
Total	**1926**	**1926**
VARP Core Program Memory		
Test Program	*Instructions*	*Bytes*
check-test	5	65
read port test	580	7540
bypass test	148	1924
false-write-back test	74	962
execution unit test[25]	-	4330
rescheduling algorithm	732	9516
static data	-	512
Total	**1539**	**24849**
Arbiter Memory		
Fault State Registers		81
Full System		
Total program and data memory overhead		26856

Table 5-7: Memory-Overhead for the adaptive diagnostic software-based self-test.

The service routine including all data structures requires less than 2 kByte of memory. Please note that the adaptation of all test programs and of the rescheduling algorithm is done in the data memory of the service core. However,

[25] Please note that the execution unit test was not included in the system configuration shown in figure 5-6. However, the memory consumption of a test program for a VARP-like execution unit was determined in [81].

this adaptation can be done separately for each instruction such that memory must be provided only for storing a single instruction.

The test programs for the VARP core require about 24 kByte of memory. The total memory consumption of 26856 bytes is independent of the size of the user application. I.e., even for very small user applications this amount of memory is needed for self-testing purposes. The size of the memory needed during the execution of the rescheduling algorithm depends on the size of the program fragments that are moved into the data memory of the VARP processor, and on the number of basic blocks in the user application. For example, the memory consumption m of the rescheduling-implementation used in [153] is bounded by

$$m \leq 1160 + 4 \cdot bb + 104 \cdot fr \text{ bytes},$$

where bb is the number of basic blocks in the user application and fr is the number of instructions in a fragment that is moved into the data memory for rescheduling purposes. In that formula fr can be selected in such a way that the required data memory for rescheduling does not exceed the available data memory.

5.6.3 Runtime of the Adaptive Diagnostic SBST

The runtime of the adaptive diagnostic SBST depends on the fault state of the VARP processor. In [153] a comprehensive VHDL model of the system from figure 5-5 was implemented. Various faults were injected in that model, and the adaptive SBST routine was executed. The runtimes are reported in table 5-8 for various fault states in microseconds.

The column *Fault State* specifies the stuck-at faults injected in the VHDL model. A fault in a register (fault states 2 to 13) is always a stuck-at fault in an arbitrary bit position of that register. Injected address faults (fault states 15 to 18) always affect a single bit-position of the register address, such that only half of the registers are accessible through the affected read port. Faults in the write ports affect the data-signals from the write-back register of a slot into the register file.

Most of the test time (around 55%) is needed for the slot test (shown in column *slot test*). This includes the time for the check-test and for the read port tests, including all adaptations made in the test code of the read port test. The adaptation times for the bypass test program (*bp*) and the false-write-back test program (*fwb*) are shown in columns *bp* and *fwb*. The execution time of both test programs is shown in column *bp/fwb-test*.

No.	Fault State	slot test	adaptation		bp/fwb test	adapt repair	Total
			bp	fwb			
1	no fault	794	0	3	78	2	877
2	fault in fetch register 1	799	70	33	77	507	1486
3	fault in fetch register 0	819	70	49	77	566	1581
4	faults in fetch registers 0 and 2	587	76	49	76	566	1354
5	faults in fetch registers 0 and 1	597	76	63	76	597	1409
6	faults in fetch registers 0 to 2	364	83	77	75	628	1227
7	faults in fetch registers 1 to 3	346	83	33	75	507	1044
8	faults in write-back registers 1 to 3	1078	83	33	75	507	1776
9	faults in write-back registers 0, 1 and 3	1091	83	63	75	597	1909
10	fault in write-back register 1	795	70	33	77	507	1482
11	fault in write-back register 3	1076	70	33	77	507	1763
12	faults in registers 35 to 42	796	0	39	78	507	1420
13	faults in r0 and r8	796	0	35	78	-	909
14	r0 not accessible through read port 0	795	0	33	78	529	1435
15	address fault in read ports 0 and 1	796	0	39	78	549	1462
16	address faults in read ports 0 to 3	795	0	36	78	547	1456
17	faults in fetch registers 0 and 1, address faults in read ports 4, 5, 6, 7	597	76	69	76	-	818
18	faults in fetch registers 0, 1, and 2, address faults in read ports 7 and 8, fault in register 19 to 26	364	83	79	75	-	601
19	faults in write ports 0 to 2	799	83	87	75	-	1044
20	fault in fetch register 0 and write port of slot 1	820	76	63	76	597	1632
Average:		**710**	**54**	**47**	**73**	**337**	**1271**

Table 5-8: Runtimes of the adaptive SBST in microseconds for various fault states of the VARP processor.

The column *adapt repair* shows the runtime needed for adapting the rescheduling-algorithm to the detected fault state. Rows without a specified time represent fault states where the self-repair routine could not be adapted to the detected fault state, because the adaption of the self-repair routine does not include a register renaming. In all cases without specified repair time a used register of the repair routine is faulty or not accessible in the functioning slots. In all cases, the execution and adaptation of the test programs takes less than two milliseconds. This is negligible compared with the repair time needed for larger user applications (see table 4-2). However, the short test time allows for the execution of the fine grained diagnostic self-test even in short execution breaks, as it was claimed in section 4.3.6 for the combination of a hybrid approach with a fine-grained off-line self-repair method.

5.6.4 Quality of the Adaptive Diagnostic SBST

The quality of the adaptive SBST is evaluated for the data path components of the VARP processor for which a self-repair method exists in terms of fault coverage, and in terms of correct fault classification. I.e., an injected fault in the structural VHDL model of the VARP processor must be detected and classified correctly according to the caused functional misbehavior. In [148] the fault coverage of the test programs and the fault classification was evaluated in a VARP processor model for the read ports, bypasses, and single registers of the register file. For this reason structural models of these components were implemented manually and integrated into the VARP processor model. All feasible single stuck-at faults were injected in these models. Table 5-9 shows the number of collapsed faults for these components and the fault coverage achieved by executing the proposed test programs on the VARP core.

Component	Faults	FC	Good Classification	False Classification
Read Port	6092	100%	6092	0
Bypass	1278	100%	1278	0
Register	566	100%	566	0

Table 5-9: Achieved fault coverage with adaptive SBST programs for various components of the VARP processor.

For all three components 100% fault coverage is achieved. The fault classification capability was evaluated manually. I.e., for each injected fault (or in most cases groups of injected faults), the expected functional misbehavior was evaluated in an analytical manner. Then it was checked whether or not the expected functional misbehavior was identified correctly from the test programs. All injected faults were classified correctly by the test programs.

Please note, that the manually developed structural models of the processor components were needed for the analysis of the fault classification, because the manually developed structural models are well understood. When using a synthesized structural model of a component for the analysis, it becomes almost impossible to understand the functional misbehavior caused by an injected fault. In other words, the mapping of structural faults to functional misbehavior was supported by the manually developed model.

However, for larger components such a method becomes impractical for evaluating the diagnostic capability of functional test programs. For this reason, Schölzel and Koal have proposed in [163] a method that allows for an automated classification of structural faults according to the caused functional misbehavior with respect to a given structural fault model. In particular, an ATPG-tool is used for test pattern generation. A synthesized model of the considered component is used as input for the ATPG-tool together with some constraints. The constraints are selected in

such a way that they force a particular functional behavior of the component. A simple example of such a constraint for a read port component is that the control signal must have a particular value r. This corresponds to the functional behavior of selecting register r with that read port. During test pattern generation, the ATPG-tool only generates test patterns that respect the given constraints. The test patterns generated with the ATPG-tool will detect all faults[26] (with respect to the structural fault model) that affect this functional misbehavior. Finally the generated test patterns are mapped into a test program, such that they can be applied to the component with a software-based self-test. However, it turned out that the manually developed test programs for the read port test, bypass test, and false-write-back test are quite efficient. The test programs needed for applying the ATPG generated test patterns are five to ten times larger than the manually developed test programs [163].

So far the quality of three single test programs was evaluated. It was shown that these test programs achieve full fault coverage and fault classification for the tested components, assuming that the proposed test programs are executed correctly without being affected from any side effects due to faults in other components. This assumption may be no longer valid, if faults occur in multiple components. For this reason, in [153] the complete adaptive self-test concept was evaluated also for randomly injected multiple faults. A coarse-grained structural model of the VARP processor was used for the evaluation. I.e., a high level description of the components was used.

Component	Number of fault sites	Fault classification
Pipeline Registers		
Fetch Register	$4 \cdot 53 = 212$	slot fault
Decode Register	$4 \cdot 118 = 472$	slot faults and read port/bypass faults
Write-Back register	$4 \cdot 46 = 184$	slot fault
Register File		
Write-Port	$4 \cdot 23 = 92$	slot fault
Register	$64 \cdot 16 = 1024$	register fault
Read Ports		
Read Ports	$8 \cdot (64 \cdot 16 + 6 + 16) = 8368$	read port fault
Bypasses		
Read Port Input	$8 \cdot 16 = 128$	read port fault
Pipeline Registers Input	$8 \cdot 128 = 1024$	bypass fault
Bypass Control Input	$8 \cdot 62 = 496$	read port and bypass faults
Bypass Output	$8 \cdot 16 = 128$	read port and bypass faults

Table 5-10: Fault sites in the coarse-grained VARP processor model.

[26] This assumes that all faults in the component were classified by ATPG-tool either as detectable or undetectable under the given constraints.

Stuck-at fault were injected only at the inputs- and outputs of these components. Table 5-10 lists the components together with the number of fault sites at inputs and outputs. Again, the fault sites were classified manually according to the caused functional misbehaviour of injected faults at these fault sites. In total 12128 fault sites exist in the coarse-grained processor model. Using the stuck-at fault model these are 24256 faults. First, a full fault simulation was done for each single stuck-at fault in order to evaluate the diagnostic capability of the slot test that encapsulates the check- and read port test. The results are shown in table 5-11.

Component	Faults	Detected Faults	Good Classification	False Classification
Slot Faults	1408	1384	1364	20
Non-Slot Faults	22848	22848	22848	0

Table 5-11: Fault coverage and fault classification achieved with the adaptive SBST for injected single stuck-at faults.

All non-slot faults in the model are detected and classified correctly. In total there are 1408 faults that must be detected and classified as slot faults, i.e., the slot containing the injected fault must be declared as faulty. 1384 out of the 1408 faults were detected as slot faults, while 24 faults were not detected as slot faults. 20 out of the 1384 detected slot faults were not classified correctly, i.e., also another slot was declared as faulty. These are in total $20 + 24 = 44$ *critical faults*, whereby each slot contains 11 of them. They are further distinguished in table 5-12.

Not detected, but do not cause misbehavior	12
Not detected as slot fault, but cause misbehavior	12
Detected, but false classification	8
Detected, but no fault handling possible	12

Table 5-12: Detailed classification of the critical faults from table 5-11.

The 12 stuck-at-0 faults in the first row (3 in each slot) affect the early generated *nop*-signal in the fetch- and decode registers. This signal is used in the execution stage for masking load-signals that are generated accidently, when the opcode of a fetched operation is changed in the pipeline. A stuck-at-0 fault of that *nop*-signal is not observable, if there is only a single stuck-at fault in a slot, but it will not cause a functional misbehavior of the processor. The 12 stuck-at-1 faults in the second row affect the *valid*-signal of a generated result. They are not detected as slot faults, but they cause some misbehavior that is detected during the bypass test. Hence they are classified as bypass faults. The 20 detected faults in rows three and four of table 5-12 (5 faults per slot) affect the opcode of a NOP before it is stored in the fetch-register. 8 out of the 20 faults change the opcode of a NOP into an operation that generates an invalid load-signal that disturbs the execution of operations in other slots. The remaining 12 faults change the opcode of a NOP into the opcode of a branch operation that will disturb the program execution. Such faults cannot be handled with the means of the software-based self-repair. A

solution to this problem would be another instruction encoding, where the opcodes of branch operations have a hamming distance larger than one from the NOP opcode. By this a single stuck-at fault in the fetch-registers will not change the opcode of a NOP into a branch operation that disturbs the execution of operations in faultless slots. From the presented analysis of the faults in table 5-12 follows that the diagnosis of single stuck-at faults fails only for 32 faults that cause a functional misbehavior. Thus, the proposed adaptive SBST provides 99.87% fault coverage for the coarse-grained data path model of the VARP processor, including a correct classification. Hence, almost all faults that affect a single component in the data path for which a self-repair mechanism is available can be detected and localized correctly. Please note that these investigations do not include the control logic, because for the control logic no software-based self-repair technique is available. Nevertheless, faults in the control logic will be detected, too, by the service core, because it expects the response from the test program (i.e., the *halt*-signal) within a specified period of time. If this signal is not received, then the control logic is considered as faulty, and the system has a failure.

With the coarse-grained VHDL-model approximately 445 fault injection simulations per hour could be performed [153]. I.e., the simulation of all 24256 single stuck-at-faults took 54 hours. There are approximately 294 million double stuck-at-faults in the coarse-grained VHDL-model. The simulation of all of them would take more than 75 years. For this reason 100.000 fault simulations were done with randomly injected faults. 50.000 fault simulations were carried out by injecting double faults, and 50.000 fault simulations were carried out by injecting 5 faults per simulation. The results are shown in table 5-13.

Fault Model	Fault Simulations	All Faults in Data Path Components	Good Classification	False Classification
2 injected SA-faults	50000	46184	45856	328
5 injected SA-faults	50000	41019	40339	680

Table 5-13: Fault coverage and fault classification achieved with the adaptive SBST for injected multiple stuck-at faults.

Please note that the fault injection campaign was not restricted to data path components of the VARP processor. I.e., faults were also injected in the control logic. However, the proposed adaptive diagnostic self-test does not account for faults in the control logic, because such faults cannot be handled by the subsequent software-based self-repair mechanisms. For this reason table 5-12 lists explicitly the number of those fault simulation runs where all injected faults are located in data-path components. In 3816 simulation runs at least one of the injected double faults affected the control logic. The remaining 46184 double faults in the data path components were all detected, and 45856 of these detected double faults were classified correctly according to the classification shown table 5-10. I.e., 328 faults were not classified correctly. 325 out of these 328 double faults contain

at least a single critical fault as it was shown in table 5-12. The three remaining false classified faults occur due to address faults, when writing and reading a register with the test programs. I.e., one fault affects the register address when writing a register, such that the wrong register is initialized. Exactly the same address fault is caused by the second fault in the read port. By this, a *check*-operation receives the value from a wrong a register, but this is not detected, because the correct value was written before into the wrong register. For example, suppose that during the read-port test a register r must be initialized with -1. But due to an address fault register $r' \neq r$ is initialized with -1. Furthermore, the tested read port has an address fault, such that always register r' is read instead of register r. Then the faulty read port is considered as functioning, and the functioning read ports are declared as faulty. Please note that this fault is later detected by the false-write-back test, but it is not classified as a fault of those slots that contain the address faults, which is a flaw of the proposed test routine.

Table 5-13 also shows the results of the fault injection of 5 faults per simulation. In 8981 cases at least one of the 5 faults affected the control logic of the processor. In the remaining 41019 simulation runs all faults were located in data path components, and all of them were detected, but in 680 simulation runs not all 5 faults were classified correctly. 644 out of these 680 faults contain at least a single critical fault. In 35 out of the remaining 36 simulations, the faults were not classified correctly, because one fault affected the register address, when writing a register and a second fault caused the same address fault in a read port. In one case three slots were faulty, and the read ports of the fourth slot contained a data fault at their outputs, such that not a single register could be read correctly. Table 5-14 summarizes the fault injection results for the coarse-grained data path model of the VARP processor. Moreover in column *simulated faults* the ration of simulated faults compared to all feasible faults is listed in percent.

Fault model	**Simulated faults**	**Correct detected and classified faults in the data path**
1 injected stuck-at-fault	100%	99.8%
2 injected stuck-at-faults	0.016%	99.3%
5 injected stuck-at-faults	$5.87 \cdot 10^{-14}$%	98.3%

Table 5-14: Fault detection and classification results for the coarse-grained data path model of the VARP processor.

In all three fault injection campaigns, very high fault coverage and correct classification could be achieved for the data path components. However, only for the single stuck-at-fault model the fault coverage can be considered as meaningful, because 100% of all single stuck-at-faults in the coarse-grained data path model were simulated, and all single stuck-at faults in the data path components were simulated at gate-level, achieving almost 100% fault coverage. However, the fault coverage of the simulation runs for two and five injected faults must be considered

very carefully, because the amount of simulated faults is very low compared with the total number of faults. Moreover, single faults inside a particular component may cause some side effects that are not covered by the fault simulation at the coarse-grained data path model. Nevertheless, many successful simulation runs have proven the fault detection and diagnostic capability of the proposed adaptive software-based self-test even in the presence of faults in multiple components. This makes this self-test concept to a viable solution for detection and diagnosis of multiple permanent faults in the data path of the VARP processor at that granularity level that is needed by the previously presented self-repair routines.

5.7 Summary

In this chapter a diagnostic adaptive SBST routine for the VARP processor was presented. Of particular importance and novelty is the adaptive capability of the SBST, which allows for adapting single test programs to previously detected fault states of the processor. By this, faults in multiple components can be detected and localized correctly in a processor-based system, even when the original test routine would fail due to the usage of a faulty component for initialization and/or capturing. This property is mandatory for systems, where only software-based methods are used for the reconfiguration of the processor, because the test programs cannot expect that the hardware is configured in such a way that it can be considered as faultless. It was shown that the adaptive self-test routine is able to cope with many situations, where multiple faults are present in the VARP core, and that the developed test programs for the read ports, bypasses, and register file achieve 100% fault coverage and correct fault classification at gate-level using the stuck-at-fault model. The test of the pipeline registers could not classify all faults correctly. However, by an exhaustive fault simulation for single stuck-at faults, it was proven that such faults are rare. More than 99% of the single stuck-at-faults in the data path of the VARP core were detected and classified correctly. Even for multiple randomly injected stuck-at faults, a correct fault classification was possible in most situations. Thereby the hardware- and software-overhead can be considered as moderate. Less than 15% hardware overhead and 26 kByte of memory are needed for providing the in-field test infrastructure for the adaptive diagnostic SBST. Almost half of the hardware overhead of this test infrastructure (the arbiter) is shared with the self-repair routine or it can be avoided completely in a multi-core system with shared data memory. Please note that the additional memory overhead of 26 kByte may increases the vulnerability of the memory to temporary and permanent faults, which has a negative impact on the overall reliability of the system. However, for several reasons this problem will be not that serious. First, in systems with large applications 26 kByte memory overhead may be negligible. Second, techniques for handling permanent and temporary faults in

memories are state of the art [106, 126]. Hence, 26 kByte of additional memory may not significantly affect the reliability of the memory if it is protected by such techniques. Third, the diagnostic self-test and self-repair is only needed during system startup. Before system startup these routines may be loaded into the program memory from a more reliable storage location (e.g., from a ROM).

Although the adaptive SBST was presented for the VARP processor, the general ideas and parts of the test flow for the adaptive SBST shown in figure 5-4 are also applicable to other statically scheduled superscalar processors. For example, the test patterns presented for the read ports and bypasses may be used in other processors, too. But it is obvious that the test programs that apply these test patterns must be adapted to the instruction set architecture of other processors. Moreover, complete test pattern sets may be replaced by other test pattern sets for targeting for other fault models. For example, test programs may be developed for testing for delay faults and can replace the test programs for stuck-at faults. Please note that changing the test programs for targeting other fault models will neither affect the self-repair routines nor the general test flow shown in figure 5-6. These routines are independent from the used structural fault model for fault detection and localization.

Also the test flow that systematically checks the components of the processor with the given test programs must be adopted for other processors. The test flow shown in figure 5-6 was developed manually by knowing the details of the VARP processor and the interdependencies between the processor components. Thus, the presented self-test concept has two flaws that were not solved so far. First, an automated or at least semi-automatic method for a systematic generation of the general test flow is missing. Second, the evaluation of the diagnostic capability of such a test flow is not solved satisfactorily, because this requires the mapping of structural faults to functional misbehavior. For combinatorial components this problem was solved in [163] with the help of ATPG-tools. However, for the pipeline register and other processor components in the VARP processor this mapping was done manually in an analytical manner, in order to have reference values for the diagnosis results. Obviously such a proceeding may be error-prone, and some functional misbehavior may be not foreseen during the manual analysis, especially for multiple faulty components. For this reason, a formal method for deriving functional misbehavior caused by multiple structural faults in various processor components would be very beneficial. Both problems provide potential for an interesting piece of work for future research.

5.8 Conclusions

Hardware-based, software-based, and hybrid methods for administrating inherently available hardware-redundancy in a statically scheduled superscalar processor were investigated in this thesis. Most of them target for handling permanent faults during the startup phase of the system. This becomes of particular interest for embedded systems that are manufactured for performance and/or power reasons with less reliable nano-scaled technologies that will suffer from permanent faults during their life time [28]. For the first time, a comprehensive software-based self-test and self-repair concept was developed, which can be used for statically scheduled superscalar processors, because in these architectures the usage of hardware resources can be controlled in software. Thereby the software-based self-repair requires a diagnostic self-test for the processor that can be executed autonomously in the field and delivers the fault state of the core at that granularity level that is needed by the self-repair approach. For this reason a diagnostic and adaptive SBST was presented in chapter 5. This test requires some additional hardware outside of the processor core for administration purposes. Thereby the overhead for the diagnostic and adaptive SBST dominates by far the overhead for the software-based self-repair. Nevertheless the total overhead is moderate for the considered processor model.

All self-repair methods – software-based, hardware-based, and hybrid methods – were demonstrated and evaluated using the VARP processor. This allows for a comparison between these concepts. The software-based self-repair methods presented in chapter 3 are characterized by avoiding any hardware-overhead in the processor core for administrative purposes. I.e., the inherently available hardware-redundancy in superscalar processors is administrated completely in software. Therefore these concepts can be used for handling permanent faults even in processors that were not designed as fault tolerant processors. Moreover, the redundancy in such processors can be used either for high performance or for high reliability, because the decision on how to use the redundancy is made in software. By implementing hardware-based reconfiguration schemes for the same processor in chapter 2, it turned out that for low failure rates the reliability of both methods did not significantly differ. However, the simple hardware-based rebinding scheme, which was used in order to maintain the hardware-overhead low, causes a strong performance degradation of approximately 100%, even for single faults. This performance degradation could be significantly reduced with software-based methods. Reducing the performance degradation of hardware-based methods is also possible with more complex hardware-based scheduling schemes. But this causes more hardware overhead, which, in turn, decreases the reliability of such an approach. Moreover, for low failure rates, handling of faults in multiple slots of the

VARP processor did not significantly improve its reliability compared with fault handling in a single slot only. But it could be shown that for high failure rates handling of multiple faults becomes favourable. Lowering the granularity of the fault handling below slot level did not significantly increase the reliability of the processor, neither for high nor for low failure rates. However, in the presence of multiple faults this reduces the performance degradation, which is caused by avoiding the usage of defective components, significantly. Hence, the software-based rescheduling at fine-grained granularity level can be considered as a powerful method for improving the reliability of statically scheduled superscalar processors affected by high failure rates without losing too much performance. Because no additional hardware is introduced in the processor core for administrating the fine-grained hardware redundancy, this benefit is also achieved without increasing the sensitivity of the processor against transient faults.

Unfortunately the software-based self-repair methods are characterized by a longer reconfiguration time during the startup phase of the system, and they cannot be used for handling permanent faults that manifest during the execution of the user application. Thus, pure software-based methods may be only used in systems where permanent fault handling only during the startup phase is acceptable or the service of the user application can be interrupted for a short period of time. If this is not the case, then the software-based methods may be supported by additional fault tolerance methods for concurrent error detection and recovery. For this reason in chapter 4 it was shown how the presented software-based methods can be used in hybrid approaches for supplementing hardware-based fault tolerance techniques. Such a combination allows for a reduction of the startup time, when the software-based methods are combined with a hardware-based reconfiguration of the processor. Moreover, the time consuming software-based methods can be used for a fine-grained reconfiguration in execution breaks or during startup, while hardware-based methods are used for fast, but coarse-grained on-line fault handling. By this the complexity of the on-line methods remains low, and after a fine-grained off-line reconfiguration all available non-faulty resources of the processor are well utilized.

In particular the presented software-based self-repair and self-test provides an alternative solution to pure hardware-based solution. Whether they can be used efficiently or not depends on the properties of the faults affecting the system, the architecture of the used processor, and the application scenario that determines the time constraints for error detection and fault handling. Even in scenarios where pure software-based methods cannot be used, they provide an interesting alternative for supplementing hardware-based methods in hybrid approaches. Especially the development of more hybrid cross-layer methods, where the software-based self-repair methods are used for fine-grained self-repair, will be a

promising strategy for overcoming the challenges of building long-living reliable processors from unreliable hardware components. The software-based techniques presented in this thesis provide a solid foundation for the further development of such cross-layer approaches.

Appendix A

VARP Instruction Set Architecture

Encoding of instructions is shown in figure 2-2. Encoding of operations is shown in the subsequent tables. Each operation is composed of the bit-fields

$$b_7 \ldots b_0 \; Src1 \; Src2 \; Sst$$

where $b_7 \ldots b_0$ is a bit-group of eight bits and *src1*, *src2*, and *dst* are bit-groups of six bits each. In the shown assembler mnemonic a, b, $c \in \{0,\ldots,63\}$ are register numbers. R$a(i)$ is the i-th bit position in register Ra. M[Ra] denotes the value in the memory that is referenced by the address stored in Ra. $k \in \{0,\ldots,2^{16}-1\}$ is a constant value, whereby $k_{15} \ldots k_0$ represent the single bits of the binary representation of k. x is used in the tables for representing don't care values. $FS[y,z]$ denotes bit z of fault state register fsr_y, whereby y, $z \in \{0,\ldots,63\}$.

Group 0: NOP, memory access operations, test operations						
Mnemonic	**Meaning**	**Encoding**				
		$b_7 \ldots b_4$	$b_3 \ldots b_0$	***Src1***	***Src2***	***Dst***
nop	no operation	0000	0000	x	x	x
ld0 [Ra],Rc	R$c \Leftarrow$ M[Ra]	0000	0001	a	x	c
ld1 [Ra],Rc		0000	0010			
st0 [Ra],Rc	M[Ra] $\Leftarrow$ Rc	0000	0011	a	c	x
st1 [Ra],Rc		0000	0100			
chk_l_0 Ra,y,z	if R$a = 0$ then FS[y,z] $\Leftarrow 0$ else FS[z,z] $\Leftarrow 1$	0000	0101	a	y	z
chk_r_0 y,Ra,z	if R$a = 0$ then FS[y,z] $\Leftarrow 0$ else FS[y,z] $\Leftarrow 1$	0000	0110	y	a	z
chk_l_1 Ra,y,z	if Ra = -1 then FS[y,z] $\Leftarrow 0$ else FS[y,z] $\Leftarrow 1$	0000	1010	a	y	z
chk_r_1 y,Ra,z	if Ra = -1 then FS[y,z] $\Leftarrow 0$ else FS[y,z] $\Leftarrow 1$	0000	1001	y	a	z
halt	Stops the program execution	0000	0111	x	x	x
ld_fs y,z,Ra	if FS[y,z] = 1 then R$a \Leftarrow 1$ else R$a \Leftarrow 0$	0000	1000	y	z	a

Group 1: Register indirect branch operations						
Mnemonic	**Meaning**	**Encoding**				
		$b_7 \ldots b_4$	$b_3 \ldots b_0$	***Src1***	***Src2***	***Dst***
jz Ra,Rb	if(Ra(0) = 1) then PC ⇐ Rb	0001	0001	b	a	x
jnz Ra,Rb	if(Ra(0) = 0) then PC ⇐ Rb	0001	0010	b	a	x
js Ra,Rb	if(Ra(1) = 1) then PC ⇐ Rb	0001	0111	b	a	x
jns Ra,Rb	if(Ra(1) = 0) then PC ⇐ Rb	0001	1000	b	a	x
jc Ra,Rb	if(Ra(2) = 1) then PC ⇐ Rb	0001	1001	b	a	x
jnc Ra,Rb	if(Ra(2) = 0) then PC ⇐ Rb	0001	1011	b	a	x
jo Ra,Rb	if(Ra(3) = 1) then PC ⇐ Rb	0001	1100	b	a	x
jno Ra,Rb	if(Ra(3) = 0) then PC ⇐ Rb	0001	1101	b	a	x
jmpt Ra,Rb	if(R$a \neq 0$) then PC ⇐ Rb	0001	0101	b	a	x
jmpf Ra,Rb	if(R$a = 0$) then PC ⇐ Rb	0001	0110	b	a	x
call Ra,Rb	Rb ⇐ PC+1; PC ⇐ Ra	0001	0011	a	x	b
jmp Ra	PC ⇐ Ra	0001	0100	a	x	x
Group 12: Arithmetic and logical operations						
cml Ra,Rc	Rc ⇐ not Ra	1100	0001	a	x	c
inc Ra,b,Rc	Rc ⇐ Ra + b	1100	0010	a	b	c
dec Ra,b,Rc	Rc ⇐ Ra – b	1100	0011	a	b	c
add Ra,Rb,Rc	Rc ⇐ Ra + Rb	1100	0100	a	b	c
adc Ra,Rb,Rc	Rc ⇐ Ra + Rb + 1	1100	0101	a	b	c
sub Ra,Rb,Rc	Rc ⇐ Ra – Rb	1100	0110	a	b	c
sbb Ra,Rb,Rc	Rc ⇐ Ra – Rb – 1	1100	0111	a	b	c
and Ra,Rb,Rc	Rc ⇐ Ra and Rb	1100	1000	a	b	c
or Ra,Rb,Rc	Rc ⇐ Ra or Rb	1100	1001	a	b	c
xor Ra,Rb,Rc	Rc ⇐ Ra xor Rb	1100	1010	a	b	c

cmp Ra,Rb,Rc	Rc ⇐ 000000000000$vcsz$, whereby
	if(Ra = Rb) then z ⇐ 1 else z ⇐ 0
	if(Ra < Rb) then c ⇐ 1 else c ⇐ 0
	if((Ra – R$b \geq 2^{15}$) then s ⇐ 1 else s ⇐ 0
	if((R$a < 2^{15}$ && R$b \geq 2^{15}$ && Ra – R$b \geq 2^{15}$) ||
	(R$a \geq 2^{15}$ && R$b < 2^{15}$ && Ra – R$b \leq 2^{15}$))
	then v ⇐ 1 else v ⇐ 0
		1100	1011	a	b	c

shr Ra,Rc	Rc ⇐ shr Ra	1100	1100	a	x	c
shl Ra,Rc	Rc ⇐ shl Ra	1100	1101	a	x	c
rrc Ra,Rc	Rc ⇐ rrc Ra	1100	1110	a	x	c
rlc Ra,Rc	Rc ⇐ rlc Ra	1100	1111	a	x	c

Groups 2 to 11, 14, 15: Operations of I-Type						
Mnemonic	**Meaning**	**Encoding**				
		$b_7 \ldots b_4$	$b_3 \ldots b_0$	***Src1***	***Src2***	***Dst***
ldc k,Ra	R$a \Leftarrow k$	1111	$k_{15} \ldots k_{12}$	$k_{11} \ldots k_6$	$k_5 \ldots k_0$	a
jmp k	PC $\Leftarrow k$	1110	$k_{15} \ldots k_{12}$	$k_{11} \ldots k_6$	$k_5 \ldots k_0$	x
jz Ra, k	if(R$a(0) = 1$) then PC $\Leftarrow k$	0010	$k_{15} \ldots k_{12}$	a	$k_{11} \ldots k_6$	$k_5 \ldots k_0$
jnz Ra, k	if(R$a(0) = 0$) then PC $\Leftarrow k$	0011	$k_{15} \ldots k_{12}$	a	$k_{11} \ldots k_6$	$k_5 \ldots k_0$
js Ra, k	if(R$a(1) = 1$) then PC $\Leftarrow k$	0100	$k_{15} \ldots k_{12}$	a	$k_{11} \ldots k_6$	$k_5 \ldots k_0$
jns Ra, k	if(R$a(1) = 0$) then PC $\Leftarrow k$	0101	$k_{15} \ldots k_{12}$	a	$k_{11} \ldots k_6$	$k_5 \ldots k_0$
jc Ra, k	if(R$a(2) = 1$) then PC $\Leftarrow k$	0110	$k_{15} \ldots k_{12}$	a	$k_{11} \ldots k_6$	$k_5 \ldots k_0$
jnc Ra, k	if(R$a(2) = 0$) then PC $\Leftarrow k$	0111	$k_{15} \ldots k_{12}$	a	$k_{11} \ldots k_6$	$k_5 \ldots k_0$
jo Ra, k	if(R$a(3) = 1$) then PC $\Leftarrow k$	1000	$k_{15} \ldots k_{12}$	a	$k_{11} \ldots k_6$	$k_5 \ldots k_0$
jno Ra, k	if(R$a(3) = 0$) then PC $\Leftarrow k$	1001	$k_{15} \ldots k_{12}$	a	$k_{11} \ldots k_6$	$k_5 \ldots k_0$
jmpt Ra, k	if(R$a \neq 0$) then PC $\Leftarrow k$	1010	$k_{15} \ldots k_{12}$	a	$k_{11} \ldots k_6$	$k_5 \ldots k_0$
jmpf Ra, k	if(R$a = 0$) then PC $\Leftarrow k$	1011	$k_{15} \ldots k_{12}$	a	$k_{11} \ldots k_6$	$k_5 \ldots k_0$
Group 13: Unused						
-	-	1101	x	x	x	x

List of Figures

List of Tables

List of Listings

References

[1] *SONY Semiconductor Quality and Reliability Handbook*: 2000. Online available at: http://www.sony.net/Products/SC-HP/tec/catalog/pdf/qr_all.pdf. Last access: 2014/1/28.

[2] *Failure Mechanisms and Models for Semiconductor Devices*. Technical Report, JEDEC Solid State Technology Association, JEP122C (2006), 2006.

[3] *ITRS Roadmap - Design*: 2010. Online available at: www.itrs.net. Last access: 2011/1/28.

[4] *Intel® Core™ i7-900 Desktop Processor Extreme Edition Series and Intel® Core™ i7-900 Desktop Processor Series - Specification Update*. Technical Report, Intel, 320836-022, 2013.

[5] J. Abella, X. Vera, O. S. Unsal et al.: *Refueling: Preventing Wire Degradation due to Electromigration*. IEEE Micro, 28(6), pp. 37-46, 2008.

[6] M. Abramovici and M. A. Breuer: *Fault diagnosis based on effect cause analysis: An introduction*. Proc. of the 17th Int. Conference on Design Automation (DAC'80), pp. 69-76, 1980.

[7] J. Aidemark, J. Vinter, P. Folkesson and J. Karlsson: *Experimental Evaluation of Time-redundant Execution for a Brake-by-wire Application*. Proc. of the International Conference on Dependable Systems and Networks (DSN'02), pp. 210-217, 2002.

[8] M. A. Alam and S. Mahapatra: *A comprehensive model of PMOS NBTI degradation*. Microelectronics Reliability, 45(1), pp. 71-81, 2005.

[9] Z. Alkhalifa, V. S. S. Nair, N. Krishnamurthy and J. A. Abraham: *Design and Evaluation of System-Level Checks for On-Line Control Flow Error Detection*. IEEE Transactions on Parallel and Distributed Systems, 10(6), pp. 627-641, 1999.

[10] R. Allen and K. Kennedy: *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann, 2002.

[11] F. Anjam and S. Wong: *Configurable Fault-Tolerance for a Configurable VLIW Processor*. 9th International Symposium on Applied Reconfigurable Computing (ARC'13), pp. 167-178, 2013.

[12] T. Austin: *DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design*. Proc. of the 32nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'99), pp. 196-207, 1999.

[13] A. Avizienis: *Arithmetic error codes: Cost and effectiveness studies for application in digital system design*. IEEE Transactions on Computers, 20(11), pp. 1322-1331, 1971.

[14] A. Avizienis, J.-C. Laprie, B. Randell and C. Landwehr: *Basic Concepts and Taxonomy of Dependable and Secure Computing*. IEEE Transactions on Dependable and Secure Computing, 1(1), pp. 11-33, 2004.

[15] A. Avizienis, J.-C. Laprie and B. Randell: *Fundamental Concepts of Dependability*. Proc. of the 3rd Information Survivability Workshop (ISW'00), pp. 7-12, 2000.

[16] A. Avizienis, J.-C. Laprie and B. Randell: *Fundamental Concepts of Dependability*. Technical Report, Newcastle University Report, CS-TR-739, 2001.

[17] P. H. Bardell and W. H. McAnney: *Self-testing of multichip logic modules*. Proc. of the International Test Conference (ITC'82), pp. 200-204, 1982.

[18] M. Bashir and L. Milor: *Modeling Low-k-Dielectric Breakdown to Determine Lifetime Requirements*. IEEE Design & Test of Computers, 26(6), pp. 18-25, 2009.

[19] A. Benso, S. Chiusano and P. Prinetto: *A Self-Repairing Execution Unit for Microprogrammed Processors*. IEEE Micro, 21(5), pp. 16-22, 2001.

[20] A. Benso, S. Chiusano, P. Prinetto, et al.: *Self-Repairing in a Micro-Programmed Processor for Dependable Applications*. 15th IEEE International Symposium on Defect and Fault-Tolerance in VLSI Systems (DFT'00), pp. 231-239, 2000.

[21] P. Bernardi, E. Sanchez, M. Schillaci, et al.: *An effective technique for minimizing the cost of processor software-based diagnosis in SoCs*. Proceedings of the Conference on Design, Automation and Test in Europe (DATE'06), pp. 412-417, 2006.

[22] P. Bernardi, E. Sanchez, M. Schillaci et al.: *An Effective Technique for the Automatic Generation of Diagnosis-Oriented Programs for Processor Cores*. IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems, pp. 570-574, 2008.

[23] D. M. Blough and A. Nicolau: *Fault Tolerance in Super-Scalar and VLIW Processors*. Proc. of the 1992 IEEE Workshop on Fault-Tolerant Parallel and Distributed Systems, pp. 193-200, 1992.

[24] C. Bolchini and F. Salice: *A Software Methodology for Detecting Hardware Faults in VLIW Data Paths*. Proc. of the 2001 IEEE Int. Symp. on Defect and Fault Tolerance in VLSI Systems (DFT'01), pp. 170-175, 2001.

[25] V. Boppana: *Fault Dictionary Compaction Using Structural and Tree-Based Techniques*. Technical Report, University of Illinois at Urban Champaign, 199660509 139, 1996.

[26] V. Boppana and W. K. Fuchs: *Fault Dictionary Compaction by the Elimination of Output Sequences*. Proc. of the International Conference on Computer Aided Design (ICCAD'95), pp. 576-579, 1994.

[27] V. Boppana, I. Hartano and K. Fuchs: *Full fault dictionary storage based on labeled tree encoding*. Proc. of the 14th VLSI Test Symposium (VTS'96), pp. 174-179, 1996.

[28] S. Borkar: *Designing reliable systems from unreliable components: the challenges of transistor variability and degradation*. IEEE Micro, 25(6), pp. 10-16, 2005.

[29] F. A. Bower, D. J. Sorin and S. Ozev: *A Mechanism for Online Diagnosis of Hard Faults in Microprocessors*. Proc. of the 38th Annual International Symposium on Microarchitecture (MICRO'05), pp. 197-208, 2005.

[30] R. W. Butler and S. C. Johnson: *Techniques for Modeling the Reliability of Fault-Tolerant Systems With the Markov State-Space Approach*. Technical Report, NASA Reference Publication 1348, 1995.

[31] P. Camurati, D. Medina, P. Prinetto and M. S. Reorda: *A diagnostic test pattern generation algorithm*. Proc. of the IEEE International Test Conference (ITC'90), pp. 52-58, 1990.

[32] Y. Cao, P. Bose and J. Tschanz: *Reliability Changes in Nano-CMOS Design (Guest Editor's Introduction)*. IEEE Design & Test of Computers, 26(6), pp. 6-7, 2009.

[33] N. P. Carter, H. Naeimi and D. S. Gardner: *Design Techniques for Cross-Layer Resilience (Invited Paper)*. Proc. of the Int. Conference on Design, Automation and Test in Europe (DATE'10), pp. 1023-1028, 2010.

[34] W. Chan and A. Orailoglu: *High-Level Synthesis of Gracefully Degradable ASICs*. Proc. of the European Design and Test Conference (ED&TC'96), pp. 50-54, 1996.

[35] K. Chandrasekar, R. Ananthachari, S. Seshadri and R. Parthasarathi: *Fault Tolerance in OpenSPARC Multicore Architecture Using Core Virtualization*. International Conference on High Performance Computing (HiPC'08), 2008.

[36] H. Y. Chang: *A Distinguishability Criterion to Selecting Efficient Diagnostic Tests*. Proc. of the Springer Joint Computer Conference, pp. 529-534, 1968.

[37] C.-H. Chen, C.-K. Wei, T.-H. Lu and H.-W. Gao: *Software-based Self-testing with multiple-level abstractions for soft processor cores*. IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 15(5), pp. 505-517, 2007.

[38] L. Chen and S. Dey: *Software-Based Diagnosis for Processors*. Proc. of the 39th annual Design Automation Conference (DAC'02), pp. 259-262, 2002.

[39] L. Chen, S. Ravi, A. Raghunathan and S. Dey: *A Scalable Software-Based Self-Test Methodology for Programmable Processors*. Proc. of the 40th annual Design Automation Conference (DAC'03), pp. 548-553, 2003.

[40] L. Chen and A. Avizienis: *N-version programming: A fault tolerance approach to reliability of software operations*. 8th Int. Symposium Fault Tolerant Computing, pp. 3-9, 1978.

[41] S.-K. Chen and W. K. Fuchs: *Compiler-Assisted Multiple Instruction Word Retry for VLIW Architectures*. IEEE Transactions on Parallel and Distributed Systems, 12(12), pp. 1293-1304, 2001.

[42] W.-T. Chen, M. Sharma, T. Rinderknecht, et al.: *Signature based diagnosis for logic BIST*. Proc. of the International Test Conference (ITC'06), pp. 1-9, 2006.

[43] Y.-Y. Chen, H. Shi-Jinn and L. Hung-Chuan: *An Integrated Fault-Tolerant Design Framework for VLIW Processors*. 18th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT'03), pp. 555-562, 2003.

[44] M.-H. Chiang, J.-N. Lin, K. Kim and C.-T. Chuang: *Random Dopant Fluctuation in Limited-Width FinFET Technologies*. IEEE Transactions on Electron Devices, 54(8), pp. 2055-2060, 2007.

[45] A. Cook, M. Elm, H.-J. Wunderlich and U. Abelein: *Structural In-Field Diagnosis for Random Logic Circuits*. Proc. of the 16th European Test Symposium (ETS'11), pp. 111-116, 2011.

[46] F. Corno, E. Sanchez, M. S. Reorda and G. Squillero: *Automatic Test Program Generation: A Case Study*. IEEE Design & Test of Computers, 21(2), pp. 102-109, 2004.

[47] J. Dahlem: *Eine platzsparende Befehlscodierung für VLIW Prozessoren*. Bachelorarbeit (BTU Cottbus), 2013.

[48] A. DeHon, N. Carter and H. Quinn: *Final Report for CCC Cross-Layer Reliability Visioning Study*. Technical Report, 2011.

[49] A. DeHon and H. Naeimi: *Seven Strategies for Tolerating Highly Defective Fabrication*. IEEE Design & Test of Computers, 22(4), pp. 306-315, 2005.

[50] A. DeHon, H. Quinn and N. P. Carter: *Vision for Cross-Layer Optimization to Address the Dual Challanges of Energy and Reliability*. Proc. of the Int. Conference on Design, Automation and Test in Europe (DATE'10), pp. 1017-1022, 2010.

[51] P. Dong, L. Fan, S. Yue, et al.: *New Latch-up Model for Deep Sub-micron Integrated Circuit*. Proc. of the Ninth IEEE International Conference on Dependable, Autonomic and Secure Computing, pp. 31-36, 2013.

[52] E. B. Eichelberger and T. W. Williams: *A logic Design Structure for LSI Testing*. Proc. of the 14th International Design Automation Conference (DAC'79), pp. 37-41, 1979.

[53] R. D. Eldred: *Test Routing Based on Symbolic Logic Statements*. Journal of the ACM, 6(1), pp. 33-36, 1959.

[54] M. Elm and H.-J. Wunderlich: *BISD: Scan-based buil-in self-diagnosis*. Proc. of Design, Automation, and Test in Europe (DATE'10), pp. 382-386, 2010.

[55] M. Engel, M. Schmoll, A. Heinig and P. Marwedel: *Unreliable yet Useful - Reliability Annotations for Cyber Physical Systems*. Proc. of the 2011 Workshop on Software Language Engineering for Cyber-physical Systems (WS4C), 2011.

[56] D. Ernst, S. Das, S. Lee et al.: *RAZOR: Circuit-Level Correction of Timing Errors for Low-power Operation*. IEEE Micro, 24(6), pp. 10-20, 2004.

[57] P. Fang, J. Tao, J. F. Chen and C. Hu: *Design in hot-carrier reliability for high performance logic applications*. Proc. of the IEEE Custom Integrated Circuits Conference, pp. 525-531, 1998.

[58] F. Ferlini, F. A. da Silva, E. A. Bezerra and D. V. Lettnin: *Non-intrusive fault tolerance in soft processors through circuit duplication*. Proc. of the 13th Latin American Test Workshop (LATW'12), pp. 1-6, 2012.

[59] J. A. Fisher: *Trace Scheduling: A Technique for global microcode compaction*. IEEE Transactions on Computers, pp. 478-490, 1981.

[60] J. A. Fisher: *Very Long Instruction Word Architectures and the ELI512*. Proc. of the Tenth Annual International Symposium on Computer Architectures (ISCA'83), pp. 140-150, 1983.

[61] M. J. Flynn: *Some Computer Organizations and Their Effectiveness*. IEEE Transactions on Computers, 21(9), pp. 948-960, 1972.

[62] M. Franklin: *A Study of Time Redundant Fault Tolerance Techniques for Superscalar Processors*. International IEEE Workshop on Defect and Fault Tolerance in VLSI Systems (DFT'95), pp. 207-215, 1995.

[63] M. Franklin: *Incorporating Fault Tolerance in Superscalar Processors*. Proceedings of the Third International Conference on High-Performance Computing (HiPC '96), pp. 301-306, 1996.

[64] B. Friedrichs: *Kanalcodierung: Grundlagen und Anwendungen in modernen Kommunikationssystemen*. Springer-Verlag, 1996.

[65] S. Funatsu, N. Wakatsuki and A. Yamada: *Designing digital circuits with easily testable consideration*. Proc. of the International Test Conference (ITC'78), pp. 98-102, 1978.

[66] J. Gaisler: *A Portable and Fault Tolerant Microprocessor Based on the SPARC V8 Architecture*. Proc. of the International Conference on Dependable Systems and Networks (DSN'02), pp. 409-415, 2002.

[67] B. A. Gieseke, R. L. Allmon, D. W. Bailey, et al.: *A 600MHz superscalar RISC microprocessor with out-of-order execution*. Proc. of the IEEE international Solid-State Circuit Conference, pp. 176-178, 1997.

[68] B. Gill, N. Seifert and V. Zia: *Comparison of alpha-particle and neutron-induced combinational and sequential logic error rates at the 32nm technology node*. Proc. of the IEEE International Reliability Physics Symposium, pp. 199-205, 2009.

[69] D. Gizopoulos, M. Psarakis, M. Hatzimihail et al.: *Systematic Software-based Self-Test for Pipelined Processors*. IEEE Transactions on Very Large Scale Integration (VLSI) Systems, pp. 1-13, 2008.

[70] P. Goel: *An implicit enumeration algorithm to generate tests for combinational logic circuits*. IEEE Transactions on Computers, C-30(3), pp. 215-222, 1981.

[71] O. Goloubeva, M. Rebaudengo, M. S. Reorda and M. Violante: *Soft-error Detection Using Control Flow Assertions*. Proc. of the 18th International Symposium on Defect and Fault-Tolerance in VLSI Systems (DFT'03), pp. 581-588, 2003.

[72] A. A. Gossett, B. W. Houghlock, M. Katoozi et al.: *Single event phenomena in atmospheric neutron environment*. IEEE Transactions on Nuclear Science, 40(6), pp. 1845-1852, 1993.

[73] T. Grüning, U. Mahlstedt and H. Koopmeiners: *DIATEST: A fast diagnsotic test pattern generator for combinational circuits*. International Conference on Computer-Aided Design (ICCAD'91), pp. 194-197, 1991.

[74] L. Guerra, M. Potkonjak and J. M. Rabaey: *High Level Synthesis Techniques for Efficient Built-In-Self-Repair*. IEEE Workshop on DFT in VLSI systems, pp. 41-48, 1993.

[75] L. Guerra, M. Potkonjak and J. M. Rabaey: *High-Level Synthesis for Reconfigurable Datapath Structures*. Proc. of the IEEE International Conference on Computer Aided Design (ICCAD'93), pp. 26-29, 1993.

[76] L. Guerra, M. Potkonjak and J. M. Rabaey: *Behavioral-Level Synthesis of Heterogenous BISR reconfigurable ASIC's*. IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 6(1), pp. 158-167, 1998.

[77] S. Habermann, R. Kothe and H. T. Vierhaus: *Built-in Self Repair by Reconfiguration of FPGAs*. 12th IEEE International On-Line Testing Symposium (IOLTS 2006), pp. 187-188, 2006.

[78] Heise News: *Wettrennen um 14-Nanometer-Technik*. 2012. Online available at: http://heise.de/-1774147. Last access: 2013/8/15.

[79] J. Henkel, L. Bauer, J. Becker, et al.: *Design and Architectures for Dependable Embedded Systems*. Proceedings of the seventh IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis (CODES+ISSS'11), pp. 69-78, 2011.

[80] J. L. Hennessy and D. A. Patterson: *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, 2003.

[81] N. Hofmann: *Entwicklung eines funktionalen Tests zur Fehlerdiagnose im Feld für einen generischen VLIW Prozessor*. Bachelorarbeit (BTU Cottbus), 2013.

[82] J. G. Holm and P. Banerjee: *Low Cost Concurrent Error Detection in a VLIW Architecture Using Replicated Instructions*. Procedings of the International Conference on Parallel Processing (ICPP), pp. 192-195, 1992.

[83] J. Hu, F. Li, V. Degalahal et al.: *Compiler-Assisted Soft Error Detection under Performance and Energy Constraints in Embedded Systems*. ACM Transactions on Embedded Computing Systems, 8(4), pp. 27:1-27:30, 2009.

[84] W.-M. W. Hwu, S. A. Mahlke, W. Y. Chen et al.: *The Superblock: An Effective Technique for VLIW and Superscalar Compilation*. The Journal of Supercomputing, (7), pp. 229-248, 1993.

[85] C. Iseli and E. Sanchez: *Spyder: A reconfigurable VLIW processor using FPGAs*. Proc. of the IEEE Workshop FPGAs for Custom Computing Machines, pp. 17-24, 1993.

[86] G. Jervan, P. Eles, Z. Peng, et al.: *Hybrid BIST Time Minimization for Core-Based Systems with STUMPS Architecture*. Proceedings of the 18th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT'03), pp. 225-232, 2003.

[87] R. Joseph: *Exploring Salvage Techniques for Multi-core Architectures*. Proceedings of the 2nd Workshop on High Performance Computing Reliability, 2006.

[88] R. Karri, K. Hogstedt and A. Orailoglu: *Computer-Aided Design of Fault-Tolerant VLSI Systems*. IEEE Design & Test of Computers, 13(3), pp. 88-96, 1996.

[89] R. Karri, K. Kim and M. Potkonjak: *Computer Aided Design of Fault-Tolerant Application Specific Programmable Processors*. IEEE Transactions on Computers, 49(11), pp. 1272-1284, 2000.

[90] T. Koal, D. Scheit and H. T. Vierhaus: *A Concept for Logic Self Repair*. Proc. of the 12th Euromicro Conference on Digital System Design / Architectures, Methods and Tools (DSD'09), pp. 621-624, 2009.

[91] T. Koal, M. Ulbricht, P. Engelke and H. T. Vierhaus: *On the feasibility of combining on-line-test and self repair for logic circuits*. 16th International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS'13), pp. 187-192, 2013.

[92] T. Koal and H. T. Vierhaus: *A software-based self-test and hardware reconfiguration solution for VLIW processors*. Proc. of the 13th International Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS'10), pp. 40-43, 2010.

[93] B. Könemann, J. Mucha and G. Zwiehoff: *Built-In Logic Block Observation Technique*. Proc. of the International Test Conference (ITC'79), pp. 37-41, 1979.

[94] I. Koren and C. M. Krishna: *Fault-Tolerant Systems*. Morgan Kaufmann, 2007.

[95] R. Kothe and H. T. Vierhaus: *Embedded Diagnostic Logic Test Exploiting Regularity*. Proc. of the 11th IEEE EUROMICRO Conference on Digital System Design Architectures, Methods and Tools (DSD'08), pp. 873-879, 2008.

[96] R. Kothe, H. T. Vierhaus, T. Coym, et al.: *Embedded Self Repair by Transistor and Gate Level Reconfiguration*. Proc. of the IEEE Workshop on Design and Diagnostics of Electronic Circuits and Systems (DDECS'06), pp. 210-215, 2006.

[97] N. Kranitis, A. Merentitis, G. Theodorou et al.: *Hybrid-SBST Methodology for Efficient Testing of Processor Cores*. IEEE Design & Test of Computers, 25(1), pp. 64-75, 2008.

[98] N. Kranitis, A. Paschalis, D. Gizopoulos and G. Xenoulis: *Software-Based Self-Testing of Embedded Processors*. IEEE Transactions on Computers, 54(4), pp. 461-475, 2005.

[99] H. Kufluoglu and M. A. Alam: *A Computational Model of NBTI and Hot Carrier Injection Time-Exponents for MOSFET Reliability*. Journal of Computational Electronic, 3(3-4), pp. 165-169, 2004.

[100] P. K. Lala: *Self-Checking and Fault Tolerant Digital Design*. Morgan Kaufmann, 2000.

[101] V. S. Lapinskii: *Algorithms for Compiler-Assisted Design-Space-Exploration of Clustered VLIW ASIP Datapaths*. Dissertation, University of Texas at Austin, 2001.

[102] J.-C. Laprie: *Dependable Computing and Fault Tolerance: Concepts and Terminology*. Proc. of the 15th International Symposium on Fault-Tolerant Computing (FTCS'85), pp. 2-11, 1985.

[103] J.-C. Laprie: *Dependable Computing: Concepts, Limits, Challenges*. Proc. of the 25th IEEE International Symposium on Fault-Tolerant Computing, pp. 42-54, 1995.

[104] J.-C. Laprie, C. Beounes and K. Kanoun: *Definition and Analysis of Hardware- and Software-Fault-Tolerant Architectures*. IEEE Computers, 23(7), pp. 39-51, 1990.

[105] A. Leininger, M. Gössel and P. Muhmenthaler: *Diagnosis of scanchains by use of a configurable signature register and error-correcting codes*. Proc. of the Int. Conference on Design, Automation and Test in Europe (DATE'04), pp. 1302-1307, 2004.

[106] J. F. Li, C. C. Yeh, R. F. Huang and C. W. Wu: *A Built-In Self Repair Scheme for Semiconductor Memories with 2-D Redundancy*. Proc. of the IEEE International Test Conference (ITC'03), pp. 393-398, 2003.

[107] J. Lienig and G. Gerke: *Electromigration-Aware Physical Design of Integrated Circuits*. Proc. of the 18th Int. Conference on VLSI Design, pp. 77-82, 2005.

[108] C. H. Lin, Y. Xie and W. Wolf: *LZW-based code compression for VLIW embedded systems*. Proc. of the International Conference on Design, Automation, and Test in Europe (DATE'04), pp. 76-81, 2004.

[109] C. A. L. Lisbôa, L. Carro, M. S. Reorda and M. Violante: *Online hardening of programs against SEUs and SETs*. Proc. of the 21th International Symposium on Defect and Fault-Tolerance in VLSI Systems (DFT'06), pp. 280-290, 2006.

[110] J.-C. Lo, S. Thanawastien, T. R. N. Rao and M. Nicolaidis: *An SFS Berger Check Prediction ALU and Its Application to Self-Checking Processor Designs*. IEEE Transactions on Computer-Aided Design, 11(4), 1992.

[111] P. G. Lowney, S. M. Freudenberger, T. J. Karzes et al.: *The Multiflow Trace Scheduling Compiler*. The Journal of Supercomputing, 7(1-2), pp. 51-142, 1993.

[112] Z. Lu, W. Huang, J. Lach, et al.: *Interconnect Lifetime Prediction under Dynamic Stress for Reliability-Aware Design*. Proc. of the IEEE/ACM International Conference on Computer Aided Design, pp. 327-334, 2004.

[113] S. A. Mahlke, D. C. Lin, W. Y. Chen, et al.: *Effective Compiler Support for Predicated Execution Using the Hyperblock*. Proc. of the MICRO, pp. 45-54, 1992.

[114] H. Mahmoodi, S. Mukhopadhyay and K. Roy: *Estimation of Delay Variations due to Random-Dopant Fluctuations in Nanoscale CMOS Circuits*. IEEE Journal of Solid-State Circuits, 40(9), pp. 1787-1796, 2005.

[115] S. R. Makar and E. J. McCluskey: *On the Testing of Multiplexers*. Proc. of the International Test Conference (ITC'88), pp. 669-679, 1988.

[116] M. Malhortra and K. S. Trivedi: *Power-Hierarchy of Dependability-Model Types*. IEEE Transactions on Reliability, 43(3), pp. 493-502, 1994.

[117] A. Masrur, P. Kindt, M. Becker, et al.: *Schedulability Analysis for Processor with Aging-Aware Autonomic Frequency Scaling*. Proc. of the 2012 IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'12), pp. 11-20, 2012.

[118] A. Meixner and D. J. Sorin: *Detouring: Translating Software to Circumvent Hard Faults in Simple Cores*. Proc. of the International Conference on Dependable Systems and Networks (DSN), pp. 80-89, 2008.

[119] S. Mitra, K. Brelsford and P. N. Sanda: *Cross-Layer Resilience Challenges: Metrics and Optimization*. Proc. of the Int. Conference on Design, Automation and Test in Europe (DATE'10), pp. 1029-1034, 2010.

[120] S. Mitra, W.-J. Huang, N. R. Saxena et al.: *Reconfigurable Architecture for Autonomous Self-Repair*. IEEE Design & Test of Computers, 23(3), pp. 228-240, 2004.

[121] S. Mitra, N. Seifert, T. M. Mak and K. S. Kim: *Soft Error Resilient System Design through Error Correction*. Proc. of the International Conference on Very Large Scale Integration (VLSI-SoC'06), pp. 332-337, 2006.

[122] S. Mitra, N. Seifert, M. Zhang et al.: *Robust System Design with Built-In Soft Error Resilience*. IEEE Computers, 38(2), pp. 43-52, 2005.

[123] S. Müller: *Rescheduling-Strategien für statisch geplante VLIW-Prozessor-Programme zur Behandlung permanenter Fehler*. Diplomarbeit (BTU Cottbus), 2010.

[124] S. Müller, M. Schölzel and H. T. Vierhaus: *Towards a Graceful Degradable Multicore-System by Hierarchical Handling of Hard Errors*. Proc. of the 21st Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP'13), pp. 302-309, 2013.

[125] Y. Nakamuro and K. Hiraki: *Highly Fault-Tolerant FPGA Processor by Degrading Strategy*. Proceedings of the 2002 Pacific Rim International Symposium on Dependable Computing (PRDC'02), pp. 75-78, 2002.

[126] M. Nikolaidis, N. Achouri and L. Anghel: *A Diversified Memory Built-In Self-Repair Approach for Nanotechnologies*. Proc. of the IEEE VLSI Test Symposium (VTS'04), pp. 313-318, 2004.

[127] E. Normand, D. L. Oberg, J. L. Wert et al.: *Single event upset and charge collection measurements using high energy protons and neutrons*. IEEE Transactions on Nuclear Science, 41(6), pp. 2203-2209, 1994.

[128] O. Novak, E. Gramatova and R. Ubar: *Handbook of Testing Electronic Systems*. Czech Technical University Publishing House, 2005.

[129] N. Oh, P. P. Shirvani and E. J. McCluskey: *Control-Flow Checking by Software Signatures*. IEEE Transactions on Reliability, 51(2), pp. 111-122, 2000.

[130] A. Orailoglu: *Microarchitectural Synthesis of Gracefully Degradable, Dynamically Reconfigurable ASICs*. International Conference on Computer Design (ICCD'96), pp. 112-117, 1996.

[131] S. T. Pantelides, L. Tsetseris, M. J. Beck et al.: *Performance, reliability, radiation effects, and aging issues in microelectronics – From atomic-scale physics to engineering-level modeling*. Solid-State Electronics, 54(9), pp. 841-848, 2010.

[132] P. Parvathala, K. Maneparambil and W. Lindsay: *FRITS - a microprocessor functional BIST method*. Proc.of the International Test Conference (ITC), pp. 590-598, 2002.

[133] J. H. Patel and L. Y. Fung: *Concurrent Error Detection in ALUs by Recomputing with Shifted Operands*. IEEE Transactions on Computers, C-31(7), pp. 589-595, 1982.

[134] J. H. Patel and L. Y. Fung: *Concurrent Error Detection in Multiply and Divide Arrays*. IEEE Transactions on Computers, C-32(4), pp. 417-422, 1983.

[135] P. Pawlowski, A. Dabrowski and M. Schölzel: *Proposal of VLIW Architecture for Application Specific Processors with Built-in-Self-Repair Facility via Variable Accuracy Arithmetic*. Proceedings of the 10th IEEE Workshop on Design & Diagnostics of Electronic Circuits & Systems (DDECS'07), pp. 313-318, 2007.

[136] P. Pfeifer, Z. Pliva, M. Schölzel, et al.: *On Performance Estimation of a Scalable VLIW Soft-Core in XILINX FPGAs*. Proc. of the International IEEE Symposium on Design and Diagnsotics of Electronic Circuits and Systems (DDECS'13), 2013.

[137] M. Pflanz, K. Walther and H. T. Vierhaus: *On-Line Error Detection Techniques for Dependable Embedded Processors*. Proc. of the Seventh Int. On-Line Testing Workshop (IOLTW'01), pp. 51-53, 2001.

[138] S. Pillai and M. F. Jacome: *Compiler-Directed ILP Extraction for Clustered VLIW/EPIC Machines: Predication, Speculation and Modulo Scheduling*. Proc. of the Design, Automation and Test in Europe (DATE'03), pp. 10422-10427, 2003.

[139] M. Pizza, L. Strigini, A. Bondavalli and F. Di Giandomenico: *Optimal Discrimination between Transient and Permanent Faults*. 3rd IEEE High-Assurance Systems Engineering Symposium (HASE'98), pp. 214-223, 1998.

[140] D. K. Pradhan: *Fault Tolerant Computing*. Prentice Hall, 1996.

[141] M. Psarakis, D. Gizopoulos, E. Sanchez and M. S. Reorda: *Microprocessor Software-based Self-Testing*. IEEE Design & Test of Computers, 27(3), pp. 4-18, 2010.

[142] J. Rajski and J. Tyszer: *Arithmetic Built-In Self-Test for Embedded Systems*. Prentice Hall PTR, 1997.

[143] B. Randell: *System Structure for Software Fault Tolerance*. IEEE Transactions on Software Engineering, 10(6), pp. 220-232, 1975.

[144] M. Rebaudengo, M. S. Reorda, M. Torchiano and M. Violante: *Soft-error Detection through Software Fault-Tolerance techniques*. Proc. of the 14th International Symposium on Defect and Fault-Tolerance in VLSI Systems (DFT'99), pp. 210-218, 1999.

[145] M. Rebaudengo, L. Sterpone, M. Violante, et al.: *Combined software and hardware techniques for the design of reliable IP processors*. 21st IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT '06), pp. 265-273, 2006.

[146] A. L. Reibmann and M. Veeraraghavan: *Reliability Modeling: An Overview for System Designers*. IEEE Computers, 24(4), pp. 49-57, 1991.

[147] J. Richman and K. R. Bowden: *The Modern Fault Dictionary*. International Test Conference (ITC'85), 1985.

[148] S. Röder: *Analyse und Weiterentwicklung eines adaptiven diagnostischen Selbsttests für VLIW-Prozessoren*. Masterarbeit (BTU Cottbus), 2013.

[149] B. F. Romanescu and D. J. Sorin: *Core Cannibalization Architecture: Improving Lifetime Chip Performance for Multicore Processors in the Presence of Hard Faults*. Proc. of the International Conference on Parallel Architectures and Compilation Techniques (PACT), pp. 43-51, 2008.

[150] J. P. Roth: *Diagnosis of Automata Failures: A Calculus and a Method*. IBM Journal of Research and Development, 10(4), pp. 278-291, 1966.

[151] D. Sabena, M. S. Reorda and L. Sterpone: *A New SBST Algorithm for Testing the Register File of VLIW Processors*. Proc. of the Int. Conference Design, Automation and Test in Europe (DATE'12), pp. 412-417, 2012.

[152] E. E. Sanchez, M. S. Reorda and A. P. Tonda: *On the functional test of Branch Prediction Units based on Branch History Table*. Proc. of the 19th International Conference on VLSI and System-on-Chip, pp. 278-283, 2011.

[153] S. Scharoba: *Evaluierung eines selbsttestenden und selbstreparierenden VLIW-basierten Prozessorsystems*. Diplomarbeit (BTU Cottbus), 2013.

[154] L. Scheick: *Testing Guideline for Single Event Gate Rupture (SEGR) of Power MOSFETs*. Technical Report, Jet Propulsion Laboratory, California Institute of Technology, JPL Publication 08-10, 2008.

[155] M. S. Schlansker and B. R. Rau: *EPIC: An Architecture for Instruction-Level Parallel Processors*. Technical Report, HP Laboratories Palo Alto, HPL-1999-111, 1999.

[156] M. S. Schlansker and B. R. Rau: *EPIC: Explicitly Parallel Instruction Computing*. IEEE Computers, 33(2), pp. 37-45, 2000.

[157] M. Schölzel: *Automatisierter Entwurf anwendungsspezifischer VLIW-Prozessoren*. Dissertation (BTU Cottbus), 2006.

[158] M. Schölzel: *A Delay Estimation of Rescheduling Schemes for Static Scheduled Processor Architectures*. Proc.of the 6th Workshop on Dependability and Fault Tolerance, pp. 117-124, 2009.

[159] M. Schölzel: *Scaling the Discrete Cosine Transformation for Fault-Tolerant Real-Time Execution*. Proc. of the International IEEE Conference on Signal Processing - Algorithms, Architectures, Arrangements, and Applications (SPA'09), pp. 19-24, 2009.

[160] M. Schölzel: *HW/SW Co-Detection of Transient and Permanent Faults with Fast Recovery in Statically Scheduled Data Paths*. Proc. of the Int. Conference on Design, Automation, and Test in Europe (DATE'10), pp. 723-728, 2010.

[161] M. Schölzel: *Software-Based Self-Repair of Statically Scheduled Superscalar Data Paths*. Proc. of the 13th IEEE International Symposium on Design & Diagnostics of Electronic Circuits & Systems (DDECS'10), pp. 66-71, 2010.

[162] M. Schölzel: *Fine-Grained Software-Based Self-Repair of VLIW Processors*. 26th IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems, pp. 41-49, 2011.

[163] M. Schölzel, T. Koal, S. Röder and H. T. Vierhaus: *Towards an Automatic Generation of Diagnostic In-Field SBST for Multiplexer-Based Processor Components*. Proc. of the 14th IEEE Latin-American Test Workshop (LATW'13), 2013.

[164] M. Schölzel, T. Koal and H. T. Vierhaus: *An Apaptive Self-Test Routine for In-Field Diagnosis of Permanent Faults in Simple RISC Cores*. Proc. of the Int. Symposium on Design and Diagnostic of Electronic Circuits and Systems (DDECS'12), pp. 312-317, 2012.

[165] M. Schölzel and S. Müller: *Combining Hardware- and Software-Based Self-Repair Methods for Statically Scheduled Data Paths*. 25th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems, pp. 90-98, 2010.

[166] M. Schölzel, P. Pawlowski and A. Dabrowski: *Self-Repair by Program Reconfiguration in VLIW Processor Architectures*. In: *Design and Test Technology for Dependable Systems-on-Chip*. IGI Global, pp. 241 - 265, 2011.

[167] J. Shen and J. A. Abraham: *Native Mode Functional Test Generation for Processors with Applications to Self Test and Design Validation*. Proc. of the International Test Conference (ITC'98), pp. 990-999, 1998.

[168] P. Shivakumar, S. W. Keckler, C. R. Moore and D. C. Burger: *Exploiting Microarchitectural Redundancy for Defect Tolerance*. 21st International Conference on Computer Design (ICCD'03), pp. 481-488, 2003.

[169] S. Shyam, K. Constantinides, S. Phadke, et al.: *Ultra Low-Cost Defect Protection for Microprocessor Pipelines*. Proceedings of the 12th International conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'06), pp. 73-82, 2006.

[170] S. Shyam, S. Phadke, B. Lui, et al.: *VOLTaiRE: Low-Cost Fault Detection Solutions for VLIW Microprocessors*. Workshop on Introspective Architecture (WISA06), 2006.

[171] D. P. Siewiorek and R. S. Swarz: *Reliable Computer Systems: Design and Evaluation*. A. K. Peters, Ltd. Natick, MA, USA, 1998.

[172] T. J. Slegel, R. M. Averill, M. A. Check et al.: *IBM's S/390 G5 Microprocessor Design*. IEEE Micro, 19(2), pp. 12-23, 1999.

[173] E. S. Sogomonyan, S. Weidling and M. Gössel: *A new method for correcting time and soft errors in combinational circuits*. Proc. of the 16th IEEE International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS'13), pp. 283-286, 2013.

[174] A. Sreeramareddy, J. G. Josiah, A. Akoglu and A. Stoica: *SCARS: Scalable Self-Configurable Architecture for Reusable Space Systems*. NASA/ESA Conference on Adaptive Hardware and Systems, pp. 204-210, 2008.

[175] J. Srinivasan, S. V. Adve, B. Bose and J. A. Rivers: *The Case for Microarchitectural Awareness of Lifetime Reliability*. Proc. of the 31st Annual International Symposium on Computer Architecture (ISCA'04), 2004.

[176] J. Srinivasan, S. V. Adve, P. Bose and J. A. Rivers: *The Impact of Technology Scaling on Lifetime Reliability*. Proc. of the International Conference on Dependable Systems and Networks (DSN'04), pp. 177-186, 2004.

[177] J. Srinivasan, S. V. Adve, P. Bose and J. A. Rivers: *Lifetime Reliability: Toward an Architectural Solution*. IEEE Micro, 25(3), pp. 70-80, 2005.

[178] R. C. Tekumalla: *On Reducing Aliasing Effects and Improving Diagnosis of Logic BIST Failures*. Proc. of the International Test Conference (ITC'03), pp. 737-744, 2003.

[179] S. M. Thatte and J. A. Abraham: *Test Generation for Microprocessors*. IEEE Transactions on Computers, 29(6), pp. 429-441, 1980.

[180] G. Theodorou, S. Chatzopoulos, N. Kranitis, et al.: *A Software-Based Self-Test methodology for on-line testing of data TLBs*. Proc. of the 17th IEEE European Test Symposium (ETS'12), 2012.

[181] R. S. Tupuri and J. A. Abraham: *A Novel Functional Test Generation Method for Processors using Commercial ATPG*. Proc. of the International Test Conference (ITC'97), pp. 743-752, 1997.

[182] M. Ulbricht: *Ein diagnostisches hybrides Testverfahren für einen VLIW Prozessor*. Masterarbeit (BTU Cottbus), 2010.

[183] M. Ulbricht, M. Schölzel, T. Koal and H. T. Vierhaus: *A New Hierarchical Built-In Self-Test with On-Chip Diagnosis for VLIW Processors*. 14th IEEE Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS'11), pp. 143-146, 2011.

[184] M. Ulbricht, H. T. Vierhaus and T. Koal: *Activity Migration in M-of-N-Systems by Means of Load-Balancing*. Proc. of the 15th Euromicro Conference on Digital System Design (DSD'12), pp. 258-263, 2012.

[185] A. J. van de Goor and O. Jansen: *Self Test for the Intel 8085*. Microprocessing and Microprogramming, 29(3), pp. 165-175, 1990.

[186] J. von Neumann: *Probabilistic logics and synthesis of reliable organisms from unreliable components*. In: *Automata Studies*. pp. 43 - 98, 1956.

[187] L.-T. Wang, C.-W. Wu and X. Wen: *Design for Testability: VLSI Test Principles and Architectures*. Morgan Kaufmann, 2006.

[188] P. Wohl, J. A. Waicukauski, S. Patel and G. Maston: *Effective diagnostics through interval unloads in a BIST environment*. Proc. of the 39th Design Automation Conference (DAC'02), pp. 249-254, 2002.

[189] J. F. Ziegler, H. W. Curtis, H. P. Muhlfeld et al.: *IBM experiments in soft fails in computer electronics (1978-1994)*. IBM Journal of Research and Development - Special issue: terrestrial cosmic rays and soft errors, 40(1), pp. 3-18, 1996.

[190] Y. Zorian and D. Gizopoulos: *Design for Yield and Reliability (Guest Editor's Introduction)*. IEEE Design & Test of Computers, 21(3), pp. 177-182, 2004.

Index